AFIT/GE/ENG/95D-15

Design and Analysis of Parallel Hierarchical Battlefield Simulation

THESIS
Conrad P. Masshardt
Capt, USAF

AFIT/GE/ENG/95D-15

19960611 027

DTIC QUALITY INSPECTED 5

Approved for pubic release; distribution unlimited

AFIT/GE/ENG/95D-15

Design and Analysis of Parallel Hierarchical Battlefield Simulation

THESIS

Presented to the Faculty of the Graduate School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science

Conrad P. Masshardt, B.S.E.E.

Capt, USAF

December 1995

DTIC QUALITY INSPECTED 5

Approved for pubic release; distribution unlimited

Acknowledgements

First I would like to thank Dr. Hartrum. His insights into parallel simulations, object oriented design techniques, and the C language helped to make RADSIM a solid product. Secondly, I would like to thank the members of my committee, Lt Col Wailes, Lt Col Hobart, and Capt Raines. Their classes provided me with knowledge of advanced concepts in operating systems, processor technology, and multiprocessing computers.

Even though he was not on my committee, I would like to extend a special thank you to Dr. Lamont who helped to me to refine my ideas and provided me with a firm foundation in parallel computing. The many hours of hard work will pay dividends well into the future.

I would also like to thank the following people for providing support and inspiration to myself and my family while here at AFIT. The class of GE/GCE 95D. Tom Rathbun, Darin Goosby, Dan "Go Pack" Zambon, Ed Parker, Eric and Kevin Lamkin, Kent Voss, and the many other people working behind the scenes to keep AFIT a leader in education and running smoothly.

Lastly and most importantly I would like to thank my family. My wife Michelle for caring for our two children as well as me without the appreciation she deserves. My daughter Lorelei for forcing me to take time off to have some fun. And my new son Lance, whom I look forward to having more time with to watch grow and do new things. I am truely a lucky person to have such a wonderful family.

Conrad P. Masshardt

Table of Contents

List of Figures

List of Tables

AFIT/GE/ENG/95D-15

Abstract

The purpose of this research is to determine if hierarchically partitioning a discrete event battlefield simulation reduces runtime and, if reduction exists, to characterize the runtime reduction given any particular partition configuration.

A hierarchical discrete event simulation of a main battle tank was constructed. Implementations were built for both a single processor and a multi-processing machine. The implementations used the Message Passing Interface to increase portability to other parallel and distributed configurations.

Three test cases were generated and run on three parallel and distributed environments, a network of Sun SparcStation 20's, a Silicon Graphics Power Challenge, and a Paragon XP/S. Three simplistic analytical models were constructed to develop the relationship between partition configurations.

The results showed that hierarchically partitioning simulations can produce speedup if a single event causes multiple reactions, and those reactions contain a significant requirement for processing. The analytic models were able to predict which partition configuration was better from two possible configurations if the runtime of the events and the probability of the events occurring were known.

Design and Analysis of

Parallel Hierarchical Battlefield Simulation

I. Background and Statement of Problem

1.1 Introduction

Increasing simulation complexity, size, and fidelity have produced a need to find better methods and machines for computer simulation. An example of one such simulation is battlefield simulation. Larger numbers of battlefield players have increased both the interaction complexity between players and size of the data space required to store the simulation state. While the number of players is increasing, the fidelity of each of the players is increasing as well. Simulation times for these systems are approaching and exceeding usable limits. The purpose of this research is to identify and characterize the utility of partitioning parallel discrete event battlefield simulation based on players for reduction of simulation runtime.

1.2 Background

Battlefield simulations involve many objects of various sorts, depending upon the requirements of the experiment. Some battlefield simulations contain many different objects and encompass an entire war. These simulations may contain different player models, terrain models, and signal transport models (environments). Simulations in this category are usually abstracted to probabilistic models due to the lengthy runtimes of higher fidelity

models(15, 14, 13). These simulations are used by analysts to predict battle outcomes for many instances of military equipment and tactics for using the equipment.

The other end of the spectrum contains very high fidelity models for only a limited number of objects. This type of simulation is usually associated with engineering models, expressly built to test new designs. A highly detailed model of an aircraft, a radar site, and the RF environment would be a testbed for many things including: RCS of the aircraft(2), detection range of the aircraft, effectiveness of aircraft electronic countermeasures(1), etc.

The difference in the two scenarios is the requirement for computing power. The first type of simulation contains a large number of objects with a small computational load, while the latter contains a small number of objects with a large computational load. Merging of the two types, a large number of high fidelity players, requires a large amount of processing power and time. However, this kind of simulation would provide analysts with more accurate data.

The Air Force Institute of Technology(AFIT) has implemented a battlefield simulation known as Battlesim(5, 32, 19). Battlesim is a parallel discrete event simulation based on large numbers of players with little interaction complexity or fidelity. Currently, the battlefield is partitioned spatially; each processor is assigned a section of the battlefield and maintains the state of all objects within its section. Battlefield sections are slightly overlapped in the sense that as an object approaches a boundary it is copied to the processor of the second section, and both processors maintain the state of the object.

Spatial partitioning has the advantage of only having to check neighboring processors for possible interactions, given sections that are bigger than detection ranges. However, one

might surmise that in a battle many objects would tend to congregate in a small number of sections while other sections may contain no objects. This leads to an imbalance among processor workloads, and hence longer runtimes.

1.3 Problem Statement

Given a simulation with only a few very complex players and a spatial partitioning, situations in which the players gather in a few sections of the battlefield could lead to poor speedup, and possibly even slow-down. The overhead of copying players from section to section would become a significant portion of the overall runtime, due solely to the spatial partitioning scheme.

The purpose of this research is to determine if hierarchically partitioning a discrete event battlefield simulation reduces runtime and, if reduction exists, to characterize the runtime reduction given any particular partitioning.

The goal of this research is to identify key factors in the runtime of hierarchically partitioned simulations by producing an analytic model of runtime for those simulations.

1.4 Scope

The specific objective of this research is two-fold: first, to develop a hierarchical simulation testbed and second, to determine the key factors in the runtime of the simulation. A hierarchical simulation of a main battle tank is constructed and used as a testbed. No attempt is made to model an actual battle tank. The components of the simulation and their function and effects on the tank are not intended to be realistic, but only representative of a possible player workload. The highest level of modeling is at the tank/environment

interface. The tank produces events for the environment to handle (launching of weapons, exhaust, fuel spills, etc.) and accepts inputs from the environment (add fuel, damage, etc.). Models of the signal environments (RF, IR, UHF, etc.) and the spatial manager are beyond the scope of this research effort and are not modeled. The simulation of the model uses the C language with object-oriented methods. The particular object oriented constructs are described in later chapters. The C language is used for the simple fact that C compilers exist on most parallel machines, while Ada and C++ compilers do not. Strict adherence to the object-oriented paradigm allows the possibility of conversion to another object-oriented language at a later date if so desired.

The parallel programming testbed uses the Message Passing Interface (MPI)(25). MPI is used instead of Parallel Virtual Machine (PVM)(27) because of better performance on some parallel machines (28). Also, MPI is available for many parallel and distributed machines, allowing a highly portable simulation. The experiments run with version 1.0.10 of the MPI libraries. No attempt is made to optimize the MPI implementation for any particular machine. The normal send and receive constructs are used, allowing the implementation of MPI to optimize the actual movement of data.

Several parallel and distributed computer systems are used. Some attempt is made to run at times when machine and network use is at a minimum; however this is not always possible. The systems are:

- **Network of Suns** The Air Force Institute of Technology's network of Sun SparcStation 20's, connected with fiber optic cable. These machines run SunOS 4.1.3.

- **Silicon Graphics Power Challenge** An eight node shared memory machine running IRIX v5.2. The machines used had 256Mb of RAM and 30MHz IP7 processors.
- **Intel Paragon XP/S** The Paragon XP/S at Wright-Patterson AFB. It consists of 352 general purpose nodes, each with 32MB of memory. The Paragon runs a subset of the OSF/1 UNIX(35).

1.5 Methodology

1.5.1 Literature Review. The first step in solving this problem is to investigate simulation and parallel computation through a search of current literature. The literature review included journal articles, theses, and information posted on the World Wide Web.

1.5.2 Simulation Design and Construction. Information found during the literature review is used to construct both a sequential and a parallel version of a hierarchical battle tank simulation using a conservative processor synchronization protocol.

1.5.3 Simulation Performance Testing and Analytic Model Design. Tests are performed on the simulation using three basic partition configurations. Metrics are collected to determine the number and type of events run and the total time of the different types of events.

1.5.4 Analytic Model Construction and Testing. A base simulation runtime model is constructed to determine the runtime of the simulation given the event runtimes, probabilities of events occurring, the partitioning of the simulation, and the total number

of events. This model is refined in an attempt to predict runtimes given random event occurrences and other partition configurations.

1.6 Outline of Thesis

Chapter II of this thesis contains background information in the research area resulting from a literature search of current research in simulation and parallel computation. Chapter III contains the simulation model design. Chapter IV contains the implementation of the design in a simulation of a main battle tank. Chapter V contains an analysis of the simulation results. Chapter VI contains the conclusions and a recommendations for future research.

II. Background and Literature Review

2.1 Introduction

Computer simulation is an important part of military modeling. The limitations of simulation and methods of attacking those limitations are discussed. One of the main limitations is the processing capability required for high fidelity modeling of a large number of objects in a simulation. Parallel computing is discussed as a method to provide the computing power required for these simulations. The discussion includes two methods for partitioning a simulation and several methods for processor synchronization.

2.2 Simulation

Simulation is the "imitative representation of the functioning of one system or process by means of the functioning of another"(34). Systems to be modeled may include many things from industrial processes, planetary motion, a new processor, and the environmental effects of pollution, to wars and battlefields. The systems used to accomplish this include everything from small scale functioning models to exercises to large-scale computer models.

Computer modeling has become more popular as a method of simulation for many reasons, but mainly for cost. For the cost of the computer system and programming time of the model, organizations can gain tremendous insight into their problem and possibly avoid spending millions of dollars on something that may not perform according to the desired requirements. Simulation and modeling have become so important to the military that in 1990 the Department of Defense identified simulation and modeling as one of twenty

technologies, "critical to ensuring long-term qualitative superiority of United States weapon systems (29)."

The results of simulations can avert costly redesign of military equipment, vehicles, and processes. Simulations can also help to save lives by testing new electronic countermeasures techniques(1) and helping war planners design better tactics(15). However, since computer simulation of a process is only as good as the computer model that describes it and as timely as the results, simulation users require high fidelity models which run within the allotted time.

2.2.1 Limitations of Simulation. As alluded to above, a simulation's usefulness is dependent on both the fidelity of the model and the timeliness of results. Often, such as in the modeling of an aircraft, the model requires the use of the "real" code (Flight program) and models of the hardware. However, sometimes either all the required programs do not fit in the space provided, cannot be compiled and run on the hardware performing the simulation, or take longer to run than on the real hardware. Also, as was previously stated, the simulation is only as useful as the timeliness of the results. If running the model of the next day's battle takes more than one day, it is useless.

2.2.2 Attacking the Limitations of Simulation. Several advances in both simulation and computing help to alleviate the major limiting areas. Advances in processor technology allow a single processor to approach and exceed 150 million floating point operations per second(12). Advances in software technology, coupled with the increase in processor capability, make it possible to emulate the hardware of real systems in a reasonable amount of time(8).

Simulation software technology has benefited from several advances in programming, including dynamic compilation of code(3), discrete event simulations and object-oriented programming techniques. Dynamic compilation of code makes it possible to run code compiled for another processor without much overhead. Discrete event simulations allow skipping of time when nothing important happens. Finally, object-oriented programming allows simulation designers to rapidly make changes to the simulations and to retest them without rewriting a lot of the code. New players can be added (instantiated) to the simulation and components can be easily interchanged.

Another advance that adds greatly to the computational power of computers is the use of more than one processor to complete a task. By adding another processor the theoretical computational power of the system has doubled. However, new problems arise due to the fact that the state of the simulation is either distributed over the processors, or can be modified by more than one processor. Synchronization schemes allow for this distribution of work and maintain the causality (proper time-ordering) of the simulation, but they also introduce overhead associated with inter-processor communication.

2.3 Parallelization Issues

The large amount of processing needed for a particular battlefield simulation can be provided by multiprocessing computers more readily than any other means(17). However, new problems are introduced including partitioning and synchronization overheads.

2.3.1 Partitioning. Partitioning is the process of breaking down the sequential simulation into many separate pieces to run on more than one processor. These pieces are

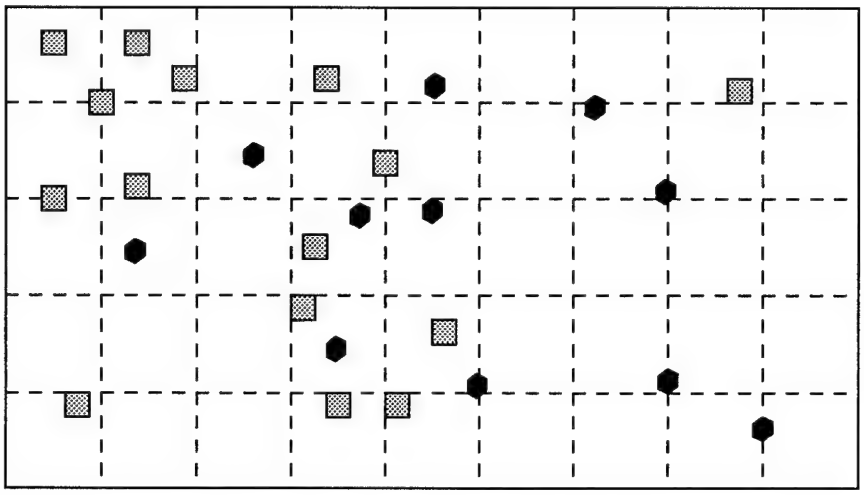

Figure 1. Possible Spatial Partitioning of a Battlefield

termed logical processes and each one represents a physical process of the simulator (16). All interactions between physical processes are modeled by time-stamped event messages sent between the corresponding logical processes (16). Research in partitioning has concentrated in two areas: spatial partitioning and hierarchical, or object-based, partitioning.

2.3.1.1 Spatial Partitioning. Spatial partitioning is based on the physical space of the system to be modeled. A good example of a spatially partitioned problem is a battlefield simulation in which the battlefield is broken down into a grid and each processor is responsible for all the objects on a particular piece of the grid (see Figure 1 for an example). Several Air Force Institute of Technology students have concentrated on a spatial partitioning for parallel discrete event simulations (19, 18, 5, 24, 32). Bergman (5) used a spatial partitioning for the battlefield simulation, *Battlesim.* The battlefield is split into sectors and as planes fly through the battlefield their "records" are transferred from one processor to another.

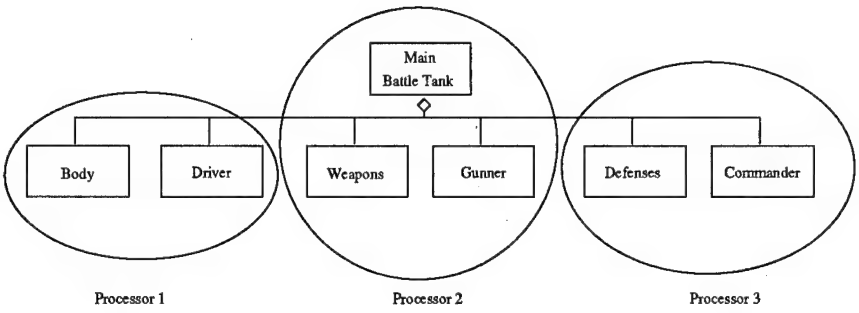

Figure 2. Possible Hierarchical/Player-based Partitioning of a Tank

2.3.1.2 Hierarchical Partitioning. The other partitioning scheme is hierarchical, or partitioning based on aggregate objects. An example of hierarchical partitioning of a battle tank is shown in Figure 2. Bain (4), Chien and Dally (11), and Zeigler (37) have completed extensive research on parallel simulation with aggregate objects. AFIT research on hierarchical partitioning has been concentrated in the area of speedup of Very High Scale Integrated Circuit (VHSIC) Hardware Description Language, or VHDL (22, 7, 20). Some research has also been completed on object partitioning of a battlefield simulation (30) and DC electrical systems (6).

Chien and Dally (11) have proposed Concurrent Aggregates as a way of implementing a hierarchical partitioning scheme. The authors explain, "Concurrent Aggregates (CA) is an object-oriented language that allows programmers to build unserialized hierarchies of abstractions by using aggregates." The concurrent aggregate model allows the concurrency of the application to remain in the simulation by allowing simultaneous messages to the aggregate components. The authors present numerous parallel simulations which were converted to run under the concurrent aggregate paradigm. The simulations include matrix multiplication, N-body simulation, printed circuit board routing, and digital logic simu-

lation. Using the concurrent aggregate concepts the authors claimed, "we can construct multiple instruction, multiple data (MIMD) programs with massive concurrency."

Bain (4) provides a formal, mathematical description of an aggregate object as the three-tuple: S, L, F, where S is a set of simple objects, L is a logical name space, and F is a function mapping L to S. Since the set S is distributed across the memories of a parallel, distributed memory system, it can provide a multi-access interface to all processors, increasing concurrency.

Zeigler's Discrete Event System Specification, or DEVS, is also a formal language specification. It is based on a hierarchical model with inputs (both internal and external), outputs, states, and state transition functions. Each model is in a particular state and inputs (events) cause transitions from one state to another.

In (36) Zeigler describes how a simulation process is managed in a parallel hierarchical system. He describes how a processor can send and receive several types of messages. One type of message signals an incoming event to the system, which contains the global time from its parent. Upon receipt of this message it updates itself and its children to the global time and responds to the parent object with the lowest-valued event it received. Ziegler explains how this hierarchical system has been modeled using PC-Scheme, a LISP dialect for microcomputers.

2.3.2 Process Synchronization. One of the most important factors of a parallel discrete event simulation is the method of synchronizing the processes involved in the simulation. Nicol (26) emphasizes the importance by stating that, "When the simulation state can be simultaneously changed by different processors, actions by one processor can

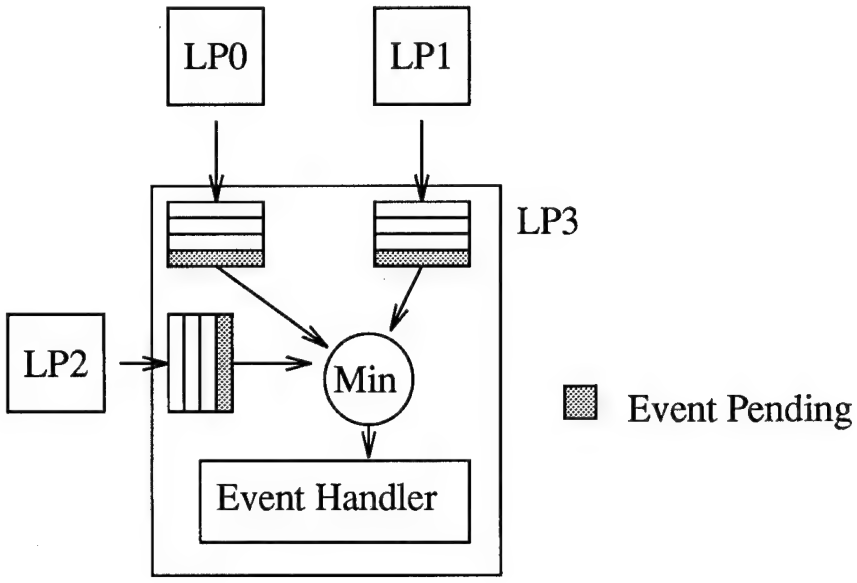

Figure 3. Conservative Synchronization of LP's

affect actions by another. One must not simulate any element (or subsystem) of the model too far ahead of any other in simulation time, to avoid the risk of having its logical past affected." He goes on to state, "Alternatively, one must be prepared to fix the logical past of any element determined to have been simulated too far." The method of synchronization generally falls into one of two categories: conservative or optimistic.

2.3.2.1 Conservative Synchronization. Fujimoto (16) states that conservative synchronization strictly avoids the possibility of a causality error ever occurring. The logical processes use some method to determine that all events that could affect the current event have been previously processed. In this manner, nothing in the future will affect the past.

Chandy and Misra (9) devised one of the first conservative protocols (see Figure 3). Their algorithm requires statically assigned communication channels between logical processes. Within the logical process each of these communication channels has an associated

input queue. Messages (time-stamped events) in each queue must arrive in the correct time sequence and are stored in First In First Out (FIFO) queues, preserving chronological ordering. The logical processes avoid causality errors by only processing an event when there is an event in every input queue and by selecting the event in the queues with the smallest time stamp.

Fujimoto (16) points out that one problem with conservative synchronization is deadlock, or failure to progress. This situation occurs when two logical processes are waiting for messages from each other. The use of Null messages is one remedy of the deadlock situation. Null messages, or messages with only a time stamp, are sent to all output channels when the logical process receives a message on one of its input channels. The null message contains the earliest time at which the process could generate a new event. This method will remove deadlock in all cases, except where cycles occur in the process graph and which have the chance to have no increment in time from the input message to the null message.

Another variation on the null message protocol is to send null messages on a "demand" basis rather than after each event (16). This reduces the number of communications between processes and thus decreases time. The protocol works by sending a request for a message on all input channels that do not have events already on them. This protocol does not suffer from the deadlock that may occur in the basic null message protocol.

Su and Seitz (31) have proposed several other null message protocols including: Eager message sending; Eager events, lazy null messages; Indefinite-lazy, single-event; Indefinite-lazy, multiple event; Demand-driven; and Demand-driven, adaptive. They determined and

noted that, "some of the variants exhibit excellent speedup over a wide range of N, limited only by the concurrency of the system being simulated."

2.3.2.2 Optimistic Synchronization. Optimistic synchronization allows logical processes to continue to process events, but uses some method to detect and correct the causality errors, usually by rolling back the simulation until it is once again safe to progress. This method requires the logical process to store its state at various times throughout the simulation run. According to Fujimoto (16), "One advantage of this approach is that it allows the simulator to exploit parallelism in situations where it is possible causality errors *might* occur, but in fact do not."

Fujimoto (16) explains that Jefferson's Time Warp mechanism detects causality errors by watching for time stamps that are smaller than the current time of the logical process. Correction resets the state of the simulation back to the simulation time when all messages were in chronological order. Resetting the simulation requires removing events that have occurred prematurely. Removing events requires resetting the previous simulation state and canceling any event messages that may have been sent to other processes as a result of the event.

Saving the state periodically satisfies the requirement to reset the state to a previous time. Anti-messages are sent to all logical processes that need to have events canceled. These processes then must go through the same rollback procedures. This continues until all processes contain chronologically correct information.

Time Warp fully corrects the errors in the simulation due to causality errors and re-runs the simulation from the point in time where events were in correct chronological

order. Lazy Cancellation attempts to simply "repair" the simulation rather than repeat the simulation fully (16). When an event is cancelled, anti-messages are held until it can be determined whether the same events would be sent. If the same events would be sent again no anti-messages are sent.

Lazy reevaluation does much the same thing, only with the simulation state rather than with the events. That is, if the new simulation state is the same as the state with the previous events, the events are not cancelled with anti-messages. Many other algorithms have been proposed for optimistic synchronization. Some of these include: Optimistic Time Windows, WOLF Calls, and Direct Cancellation (16).

2.3.2.3 Hybrid Synchronization. Fujimoto (16) notes the emergence of a third category of synchronization of discrete event simulations. These algorithms use properties of both the conservative and optimistic synchronizations. They process events like a conservative protocol, but have a limited rollback ability. If a process does not have an event on a channel it guesses what the next event time would be, and begins computations. All of the events generated by the "guessing" process are held until the actual event arrives on the channel.

2.4 Summary

Parallel discrete event simulation can achieve a significant speedup over the sequential program versions. Several methods currently exist to partition the problem across many processors and to synchronize processes once the simulation has started. Research must continue to progress in this area as new computers with new capabilities change which

algorithms are the most efficient for a machine. What may work very well for a Paragon XP/S, may not work very well for a Silicon Graphics Power Challenge Array. The challenge is for the simulation community to continue to study and propose efficient algorithms for many different computer systems. This chapter reviewed current research for parallel and distributed partitioning and synchronization algorithms.

III. Simulation Model Design

3.1 Introduction

This chapter describes the basic design philosophy for the simulation. The simulation is an object oriented, discrete event simulation built on a tree of aggregate event queues. The only communication allowed is between an object and its parent and child objects. The simulation control algorithm is described and an example is given. Parallel versions of the simulation are controlled through a specialized conservative algorithm using the *Message Passing Interface* communication libraries. Partitioning of the parallel simulation is based on aggregate objects. Each partition has a single *partition-parent object*, which must be created for partitions which contain objects that do not have a common parent.

3.2 Simulation Model

3.2.1 Model Representation. The first step in the design process is to decompose the system. An object-oriented simulation decomposition was chosen over a functional decomposition for several reasons. The reasons include easier configuration, better reuse and maintainability, and an easier method to implement the partitioning algorithm. An object oriented approach provides better opportunity for component reuse. Maintainability also increases in object oriented decompositions because of the localization of methods to alter data. Finally, an object oriented approach leads to a hierarchical structure for complex objects. This provides a convenient basis for partitioning the simulation for parallel and distributed machines.

3.2.2 Simulation System Issues. The second step is to decide on a method to increment the time. The choices are between discrete event simulation and time-stepped simulation. Discrete event simulation was chosen because it is generally accepted to have better performance than time driven simulations, and because of the general approach followed in AFIT research. Time was chosen to be a floating point number representing the number of seconds. The next event queue was chosen to be a linear linked list with an insertion sort, based on the size of the data structure and ease of implementation.

3.2.3 Communication Between Elements, Assemblies, and the Player. In a hierarchical aggregate decomposition, communications are allowed only between a component and its parent, or a component and its children. This method was chosen over child-to-child direct communication for several reasons, two of which are the ease of implementation and initialization. All communication paths are established at initialization time and are the minimum required for any hierarchical structure. This provides a common interface for each component. Design of components requires less work because only the parent and children can talk to a specific component. Therefore, the component needs only to know the type of the children and parent, not any of the other components. Debugging is also easier for the same reasons. All communication with the environment must occur through the top object in the hierarchy.

3.2.4 Simulation Control Structure. Simulation control is based on the hierarchical structure of the simulation and can be thought of as a tree of queues. Each object within the structure contains a time-ordered next event queue. Each object queue contains the events it has generated in the previous update cycle in addition to the first event from

each of its children from the previous cycle. The simulation progresses by continuously getting the top event from the simulation next event queue and updating the simulation to the time of that event. This process continues until the *Stop_Simulation* event is processed or when the simulation time reaches the maximum time.

Updating the simulation to a specific time can be thought of as a wave traveling down the hierarchy, bouncing back up and down any number of times before returning up the hierarchy to the top level simulation object. The wave traveling down the hierarchy represents updating child objects. Once the updates travel to the leaf nodes, the leaf nodes perform the required calculations and return with the lowest time event from their queue, represented by the wave traveling up the hierarchy.

Fluctuations in the smooth up and down motion of a wave through the hierarchy are representative of multiple events occurring with the same time as the update time. This situation occurs with simultaneous events and with a single event causing reactions in multiple child objects. For instance, consider an engine and a driveshaft. An increase in throttle causes an increase in the RPM output of the engine. This output is simultaneously transferred to the driveshaft and from there to the transmission.

3.2.4.1 Object Update. Updating the simulation to a specific time requires processing all events from the object's queue with a time less than or equal to the update time. Processing the events is done within the object's event handler. The top level object must have either a set of instructions to handle the event or a method to determine which child(children) should receive the event. Both possibilities exist within the simulation. Some events are known to all objects within the aggregate chain, allowing event handler

functions to be written directly into the objects. For instance, if an *update position* event arrives at the top level of the tank player from the spatial manager, the event is passed down through the body and the drivetrain to the treads. The treads calculate the new position based on the current position and the distance traveled by both tracks. Each of the three objects, the body, the drivetrain, and the treads all have a specific set of instructions to handle the *update position* event.

However, not all events need to be present in the event handler. If the event is unknown, the handler uses the event's originator field to determine which child object to update. Unknown events occur in an object when a child schedules a future event which no other objects need to react to. The end of the track distance updates is an example. As the tracks update their distance they schedule an event for a short time in the future. This event signals that both tracks have completed their distance traveled updates and the new position of the tank body can be calculated based on those distances.

An example of the simulation control is illustrated in the sequence of figures below. Figure 4 shows the simulation state during initialization. Each object adds the end simulation event at time infinity. The first event is passed up to the parent. Figure 5 shows the state of the simulation at the end of initialization. The child objects have passed the first event in their queues to their parent. Figure 6 shows the top object updating to the first event in its queue (time 3). Figure 7 shows the result of this update, a message being generated by object e for object d at time 5. Figure 8 shows the top parent updating the simulation to time 5.

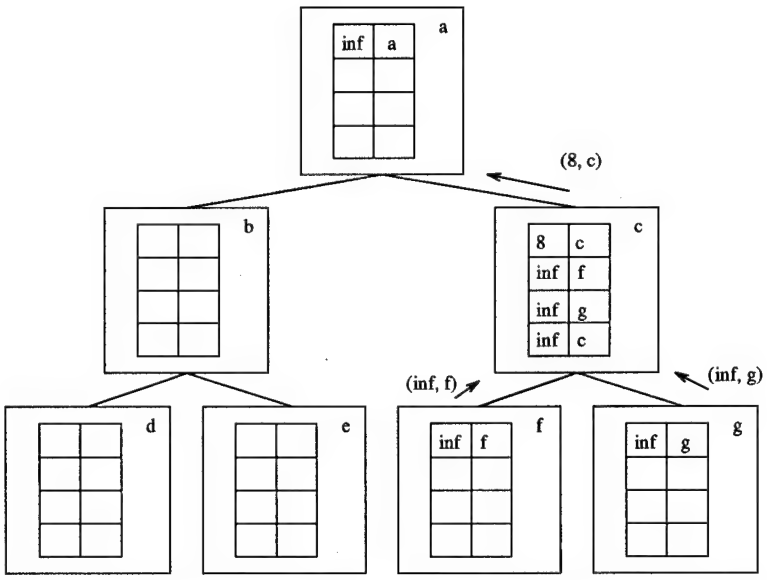

Figure 4. State During Initialization

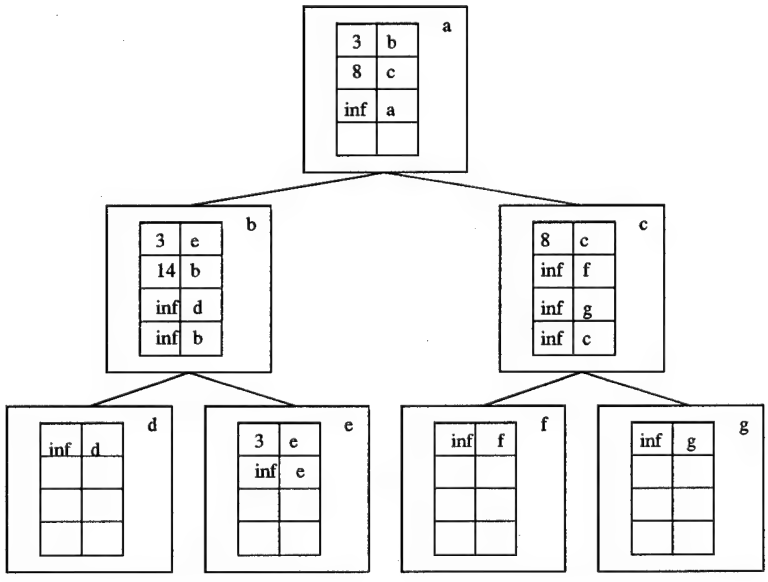

Figure 5. State After Initialization

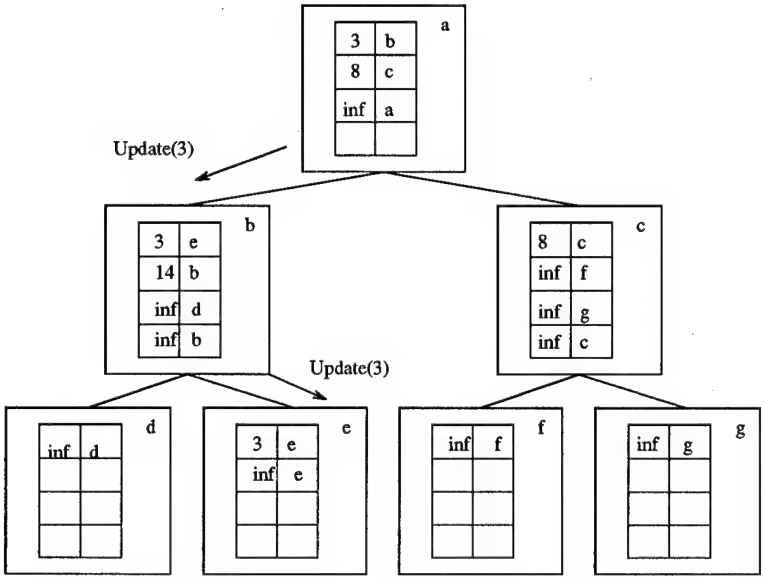

Figure 6. Updating State to Time Three

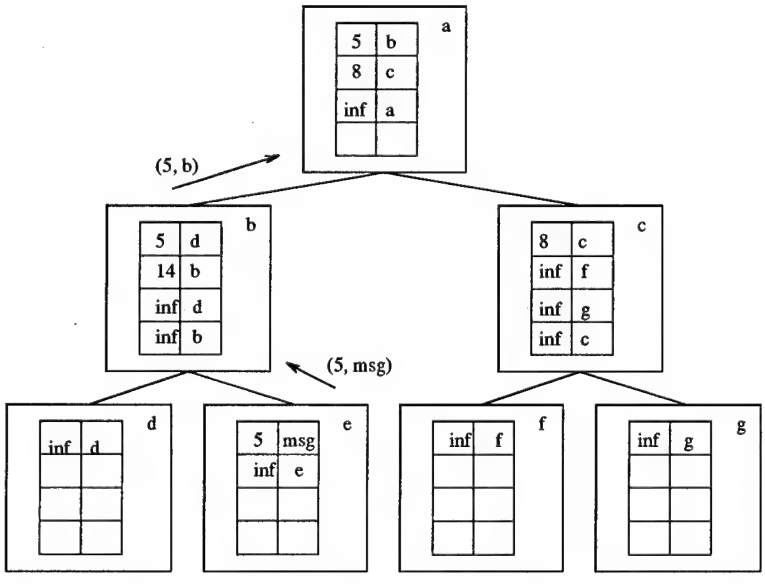

Figure 7. After Updating

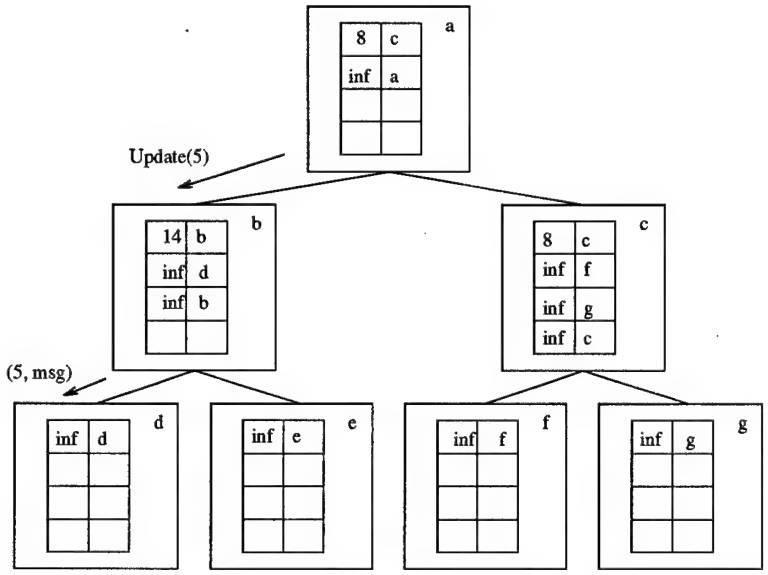

Figure 8. Updating State to Time Five

3.2.5 Initialization Files. There is an initialization file for each object instead of one single file for the simulation. The use of many files allows the simultaneous initialization of all objects and aids in readability of the simulation. Any component can initialize itself by reading the initialization file for its instance. Readability of the simulation is enhanced due to the fact that the configuration of a particular object is readily identifiable and its location known. A single large file would require the user to search through the file to find the particular object initialization information of interest.

3.2.6 Portability Issues. Every effort was made to keep the simulation design machine independent. C was chosen as the programming language for two very important reasons: compiler availability and the libraries which currently exist. Ada95 would have been the language of choice had compilers and the required libraries been available at the time of writing the simulation.

C was not specifically designed for object oriented programming. Therefore the following rules were defined to enforce an object oriented design.

- **Object Method Access** - Two possibilities exist to allow access to methods, function pointers and unique naming. A unique naming scheme was chosen. This method requires a small designator to be prepended to an object's methods to make the name unique for all objects. For example, a transmitter object could use *xtr* as a designator. The transmitter's *Create* method becomes *xtrCreate()*.
- **Inheritance** - All objects must contain several common data items and methods. The methods include *Create* and *Destroy* methods to create and destroy an instance of the object as well as others. The method name follows the naming constructs described above to avoid naming conflicts. The implementation of inheritance with the C language is further described in Section 4.3.1.1, which describes the behavior of the object oriented C code generator.
- **Data Hiding** - Access to the data of the object must be restricted to the appropriate *Get* or *Put* method. However, not all data within an object must contain either or both of these methods. To further restrict knowledge of the data structure, each *Create* method must return a void pointer to the data structure associated with the object.

3.3 Parallel Simulation Issues

One of the first design issues for any code written for a parallel machine is the programming language and availability of compilers. This is another reason C was chosen

over Ada and C++. The other prominent parallel design issues are the partitioning and synchronization algorithms and the use of a process and communication library.

3.3.1 Partitioning. The partitioning method chosen was an object based partitioning or one based on the hierarchical structure of the simulation rather than a spatial partitioning of the battlefield. This type of partitioning was chosen for two reasons: speed and ease of implementation with an object oriented design. Several possible partition configurations are shown below in Figures 9, 10, 11. All partitions must contain only a single top-level object, called the *partition-parent object.* Hence, building partitions with objects from different parts of the hierarchy requires construction of a new partition parent object with the objects contained in the partition as children of the partition-parent object. Figure 12 shows how the partition-parent object would be built for several objects that don't have a common parent as the top object in the partition.

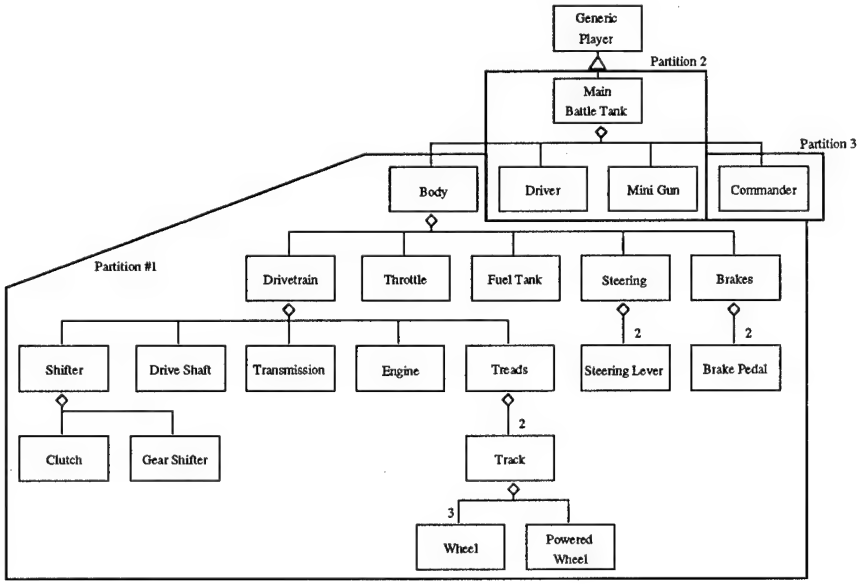

Figure 9. Partition Configuration A

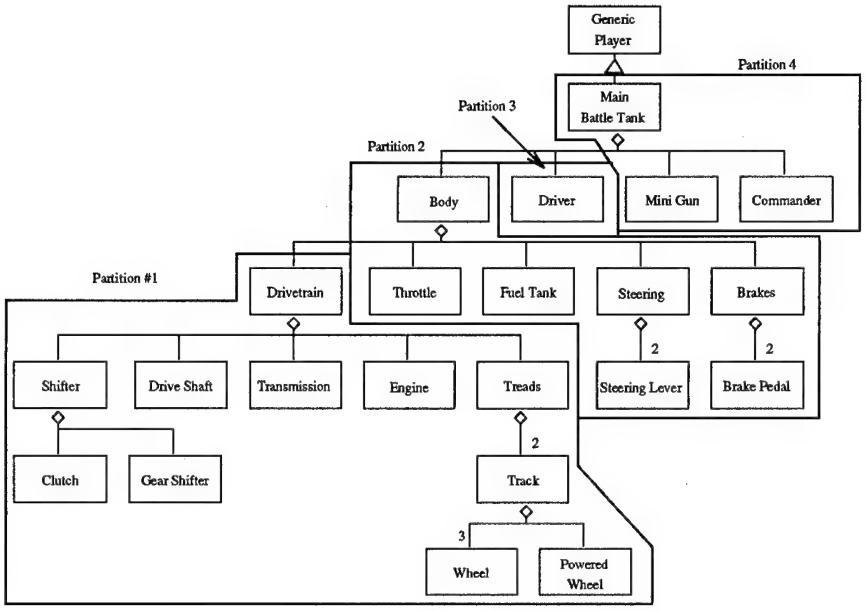

Figure 10. Partition Configuration B

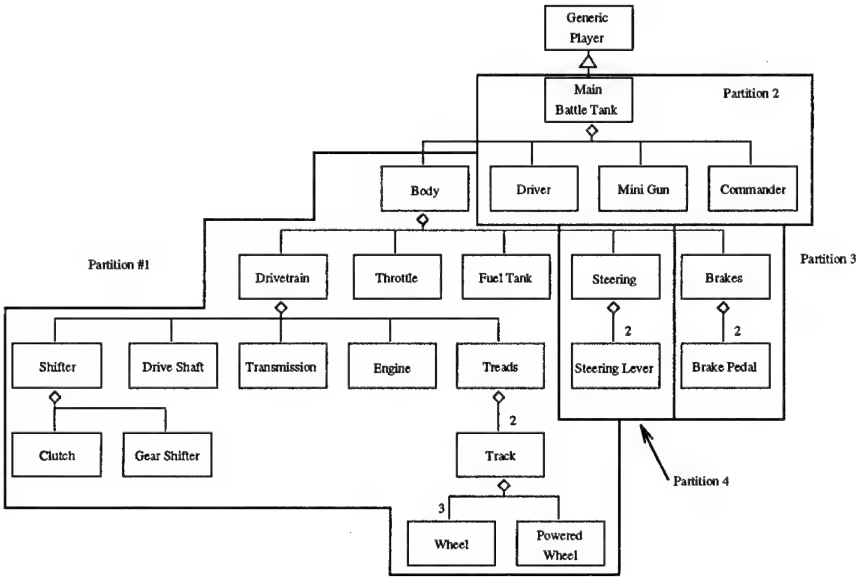

Figure 11. Partition Configuration C

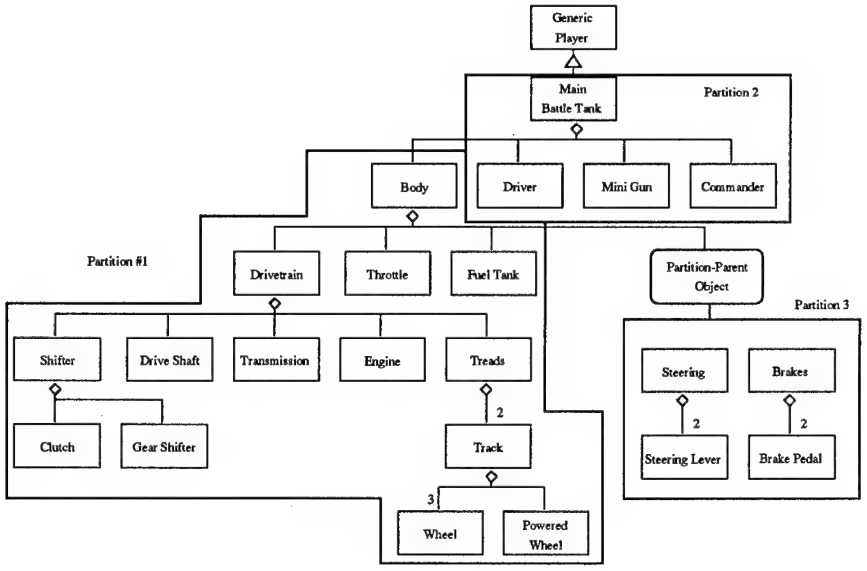

Figure 12. Partition Configuration D

3.3.2 Processor Synchronization. A form of conservative processor synchronization was chosen for the simulation. Each processor contains a single partition. The processor processes events for the partition only after it receives an event from the parent of the partition. Processing the events may require sending events to lower level (child) partitions. The parent partition waits until it has all outstanding events from its child partitions before sending the lowest time event to its parent. This is a conservative algorithm because each processor waits until it has the next event from either its parent or its child partitions before processing an event. No NULL messages are required as in most deadlock-free conservative algorithms because a parent partition and a child partition will never be simultaneously waiting on each other.

A conservative synchronization was chosen for three main reasons:

1. AFIT's research thrust is conservative synchronization.

2. Simulation state is always correct - The simulation can be stopped at any point and restarted by storing and reloading the current simulation state.

3. Least amount of storage space required - No past states are kept, only the current state is maintained.

3.3.3 Parallel Portability, Communication/Process Libraries. Portability to a large number of computer systems is highly desirable in any simulation. In addition to using a common language such as C, portability can also be enhanced by any one of several process and communication libraries such as *Parallel Virtual Machine* and *Message Passing Interface* .

3.3.3.1 Parallel Virtual Machine (PVM). PVM is a library that allows a common interface to parallel computers, homogeneous networks of workstations, and heterogeneous networks of workstations. By writing one set of code with PVM constructs the user can write a program which has the ability to run on any and all machines with a PVM implementation. PVM abstracts away the hardware, operating system level calls, and communication calls into one common interface. PVM has the ability to spawn different tasks on each logical processor in the virtual machine. By calling other PVM functions, data can be packed into messages and sent in a variety of ways to the receiver, which calls unpack routines to retrieve the message.

Any layer of abstraction is going to add overhead. Therefore, it is expected that the runtime will not be as low as with use of communication and tasking libraries provided with the machine by the manufacturer. The tradeoff is between the portability afforded

with a common interface vs the speed achieved with libraries optimized for a particular machine.

3.3.3.2 Message Passing Interface (MPI). MPI is a library similar to PVM. It also provides a common interface to the hardware and operating system through a layer of abstraction. However, this library currently only allows duplication of the same task on all processors. The program must use internal decision logic based on logical processor number to perform different actions. The affect of duplicating a single program is that more time must go into the design of the partitioning algorithm.

The Message Passing Interface (MPI) was chosen as the parallel/distributed communication/process abstraction layer for two reasons.

1. **Machine Availability/Portability** - MPI is freely available for many parallel machines, homogeneous networks of workstations, and even heterogeneous networks of workstations.
2. **Speed** - A study showed that for certain functions the MPI communication libraries are **faster** than the MPL libraries supplied by the IBM for the SP2(28). The same study also showed that MPI outperforms PVM on the IBM SP2 for most functions.

3.4 Summary

This chapter describes the design choices for the simulation. The simulation is a discrete event, object oriented hierarchical simulation which can be thought of as a hierarchical tree of event queues. The simulation runs by continually removing the first event from the simulation event queue and updating the simulation to the time of that event.

The algorithm for updating the simulation is given in the chapter as well as an example. Object oriented rules are levied on the simulation component designer to maintain an object oriented approach while using C as a programming language. The MPI library is used as the communication/process library to promote portability.

IV. Simulation Model Implementation and Analysis

4.1 Introduction

This chapter discusses the implementation of both the sequential and parallel designs of a hierarchical simulation of a main battle tank. The chapter begins with description of an object oriented C code generator, RADGEN (ConRAD's GENerator), which was developed to perform the tedious task of generating both the header and C files for an object. The remainder of the chapter is devoted to describing the implementation of a hierarchical discrete event simulation of a main battle tank, RADSIM (ConRAD's SIMulation). The simulation hierarchy is given along with descriptions of the components. The implementation details are given for the sequential algorithm. Finally, the changes that were necessary to transform the sequential version into the parallel version are described.

4.2 **RADGEN:** *An Object Oriented C Code Generator*

4.2.1 Justification. The decision to use an object oriented approach with the C language simplified the overall design of simulation, but increased the total amount of code required for a simulation. Each component in the simulation requires many methods (*Create, Destroy, Initialize, Add_Event, Update, Damage,* Get_i, Set_i, *etc.*). Each component could be constructed by copying a finished component document and changing the names to the appropriate new names within the document.

However, a second option exists: write a C code generator to transform a description file into object oriented C. This option was chosen due to the number of components that needed to be created to build the simulation. Also, it allows for future expansion and

modifications without much work. The code generator, as well as the simulation, was built with object oriented C . In fact, the *list* component was used in both the code generator and the simulation, demonstrating the reuse capability of the components!

4.2.2 Description. The *RADGEN* code generator converts object description files into an object oriented C code file and a header file. The header file lists accessible methods for the object and the C file contains the data, the methods for accessing that data, and other internal methods. Descriptions of the required files and use of RADGEN are given in the *RADGEN User's Guide*, Appendix B. Several example description files are located in Appendix C.

4.3 **RADSIM:** *A Hierarchical Simulation*

RADSIM is a hierarchical simulation of a main battle tank. The tank is composed of four main components: a body, a driver, a commander, and a minigun. The driver, commander, and minigun are not decomposed due to time constraints on the project development cycle. The body of the tank is decomposed into seventeen other components. Since this is not an accurate model of a tank, each object invokes a variable number of spin loops to vary the computational load of the objects. The terminology in the following sections is based on Joint Modeling and Simulation System (JMASS)(21) standards. A *component* refers to any and all objects in the simulation. *Elements* are leaf-node objects. A *player* is an aggregate of the simulation component. All other objects are *assemblies.* A top-down decomposition of each level of the simulation is described in the following sections.

4.3.1 Simulation Level. The simulation level (see Figure 13) contains all players in the simulation. This includes the various signal environments (RF, IR, UHF, etc), the spatial manager, and the players. However, the signal environments and the spatial manager are beyond the scope of this research effort and are not discussed further. The only player modeled is a battle tank.

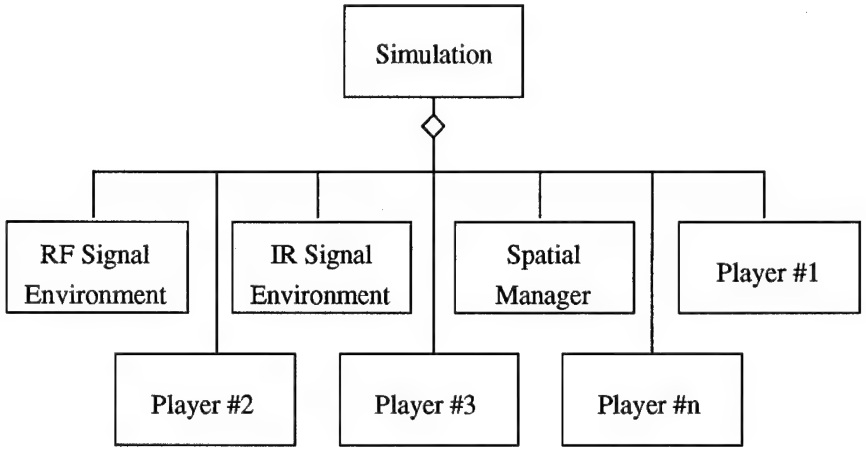

Figure 13. Simulation Layer Hierarchy

4.3.1.1 Component Inheritance. All components at and below the player level inherit some required data items (Table 1) and visible methods (Table 2). The component writer must either physically write these items into their code or use the RADGEN code generator which enters these items automatically. Since pointers cannot be transferred from one processor to another, the pointer for objects in another partition is a logical pointer to the processor number controlling the partition.

4.3.2 Player Level Components.

4.3.2.1 Battle Tank. The battle tank (Figure 14) is composed of a body, driver, mini-gun, and a commander. Omitted components include a gunner, a main gun,

Table 1. Inherited data items for each component

name	kind	Meaning
ParentNEQ	Pointer	Pointer to parent's next event queue
NEQ	Pointer	Object's next event queue
me	Pointer	Self-referential pointer
currentTime	timeType	Object's simulation time
damageLevel	3D Float	Object's current and max damage, increment level
status	integer	Status level of the object; larger = worse

and others. The tank player interfaces with the simulation to accept and provide events from and to other players in the simulation. The events the tank accepts and generates through this interface are listed in Table 3 and Table 4 respectively. The tables give the event name and a small description of each event. Since none of the components other than the battle tank are modeled, no incoming events are ever received, and the generation of outgoing messages is commented out. Any change in tank controls results in updates to the position (x,y,z), velocity, heading , and rotational velocity. At present time these events are all separate events in order to reduce the time required to construct the MPI implementation.

Notation. The notation used in the table is as follows:

- (X|Y|Z)name $\equiv \exists i \mid i \in$ {Xname, Yname, Zname}
- The "-" in the parameter field signifies a binary signal

4.3.3 Assembly Level Components.

- **Body** The body is responsible for the position and motion of the tank. The body is composed of five components: a drivetrain, throttle, fuel tank, steering, and brakes.

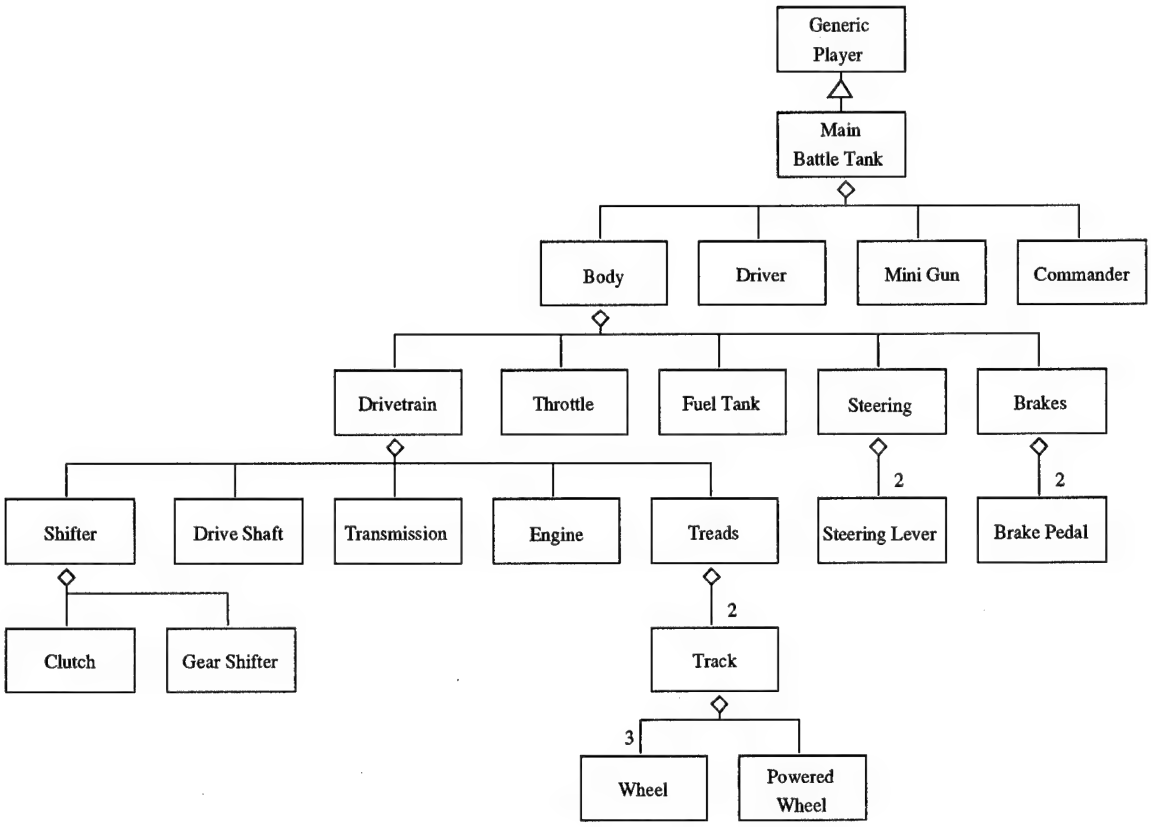

Figure 14. Battle Tank (Player) Hierarchy

Table 2. Inherited methods for each component

name	parameters	kind	Meaning
Create()		void pointer	Create object, return a void pointer to it (obj)
Init()		void	Initialize the object
	obj	objectType	A pointer to the object
	File name	string	The initialization file name
Destroy()		void	Destroy the object
	obj	objectType	A pointer to the object
AddEvent()		void	Add an event to the object's NEQ
	obj	objectType	A pointer to the object
	kind	integer	The event kind
	time	time type	The occurrence time of the event
	value	double	The value associated with the event
	origin	void pointer	The event's originator
Update()		void	Update the object
	obj	objectType	A pointer to the object
	time	time type	Time to process up to
SetparentNEQ()		void	Set the parent's NEQ pointer
	obj	objectType	A pointer to the object
	parent queue	void pointer	Pointer to the parent's NEQ
Damage()		void	Damage the object
	obj	objectType	A pointer to the object
	parent queue	void pointer	Pointer to the parent's NEQ
Setme()		void	Set the referential object pointer
	obj	objectType	A pointer to the object
	me	object pointer	Pointer to the object itself

These components contain all methods for determining position and controlling motion of the tank. The body accepts the control inputs listed in Table 5 as well as the *positionUpdate* event, which returns the position of the tank at the specified time.

- **Brakes** The brakes are responsible for putting a dampening force on the velocity of the wheels. There are two brake pedals (left and right) in the braking system which control the force applied to the left and right powered wheels, respectively. The brakes are activated on the *kchangeRLevel* and *kchangeLLevel* events. These events

Table 3. Events Accepted by the Tank Player Component from the Simulation Level Component

Event	Meaning
kupdatePosition	Request for position update
kstartInFlow	Start refueling
kstopInFlow	Stop refueling
kchangeInFlow	Change the refueling flow rate
kloadShells	Load the mini gun with ammunition

Table 4. Events Generated by the Tank Player Component for the Simulation Level Component

Event	Meaning
kdamageOthers	Damage other players
kshellFlying	Mini gun shell has left the barrel
kstart(X|Y| Z)Loc	Starting x, y, or z location of shell
kinitVel(X|Y|Z)	Starting x, y, or z velocity of shell
kspillFuel	Spill fuel into the environment
(x|y|z)positionUpdate	New x, y, or z position
velocityUpdate	new tank velocity
rotVelUpdate	new tank rotational velocity
headingUpdate	new tank heading

contain the percentage of the maximum braking force applied to the brakes. They produce *kchangeRBrakeLevel* and *kchangeLBrakeLevel* events, respectively, which are translated into reverse revolutions in the powered wheel component.

- **Drivetrain** The drivetrain is responsible for accepting inputs to the power control of the tank (engine, gears, etc.) and providing the resulting motion of the tank. The drivetrain is composed of an engine, driveshaft, transmission, shifter assembly, and a tread assembly. The engine is controlled through the *kengStart* and *kkill* events and the amount of fuel flowing into the engine. When the engine is running, a new RPM value is produced for every change in fuel flow. The RPM value is passed to the

Table 5. Body Component Control Events

Event	Parameter	Meaning
kkill	-	Kill the engine
kchange(R|L)Position	% pushed	Change right or left steering lever position
kpressClutch	-	Press the clutch
kreleaseClutch	-	Release the clutch
kgear	the gear	Change the gear
kchangeThrottle	% open	Press or release the throttle
kchange(R|L)Level	% pushed	Change the right or left braking level
kstartEngine	-	Driver presses the starter
kstartInFlow	flow rate ($\frac{gal}{s}$)	Start refueling
kstopInFlow	-	Stop refueling
kchangeInFlow	flow rate ($\frac{gal}{s}$)	Change the refueling flow rate

driveshaft, which transfers it to the transmission. The clutch and gearshifter both provide inputs to the transmission as well. The transmission takes these three values (RPM, coasting, and gear) and produces an output RPM to the tread assembly.

- **Steering** The steering control unit is composed of two control levers. The levers can be pressed forward independently to control the forward motion of their respective tracks. Since reverse motion of the tracks is not allowed in the current model, the position of the steering levers must be between zero and one. The steering lever provides a multiplier between the output of the transmission and the treads. That is, if all conditions are correct and there is a positive output from the transmission, the forward velocity of the treads can still be zero if both steering levers are not pushed forward at all.

- **Track** A track is composed of three non-powered wheels and a single powered wheel. The actual tracks themselves are not modeled, only assumed to be in place. The

forward velocity of the powered wheel is automatically copied to the three non-powered wheels, simulating what would happen if a track existed.

- **Tread** The tread is a single unit that contains the two tank tracks. The tread is the location of the location-determination algorithm. It stores the current tank location and updates the location based on the distance of each of the two tracks has moved since the last update. The location of the tank is the location of the center of the tank. The new location is determined by calculating the arc produced by the difference in distance between the two treads and locating the (x,y,z) location of the endpoint of that arc.

4.3.4 Element Level Components.

- **Brake Pedal** This is the control point for the braking function. The brakes receive a number representing the percentage the pedal is pushed down and produce a braking force (in $\frac{reverse\ revolutions}{sec}$) based on total braking force possible. This value is sent to the tracks which use it in conjunction with their forward revolutions to produce a forward distance traveled.

- **Clutch** Depressing the clutch causes the neutral signal to be sent to the transmission, which in turn causes the tank to "coast".

- **Commander** The commander controls the mini-gun. Commanders move the mini-gun and shoot at targets as programmed. The current algorithm to shoot is: every time a position update is received, the commander raises or lowers the gun and fires one round.

- **Driver** The driver contains the algorithm to control the motion of the tank. Drivers are responsible for reading the route from the route point file and keeping the tank on course by making changes to the tank's control points based on location and motion updates. The current algorithm for keeping the tank within the route is described below:

    1. Upon receipt of an update: check the current location vs the next route point.
    2. If at the location, stop the tank and get the next route point and departure time
    3. Wait until the departure time has arrived.
    4. Turn the tank towards the next route point.
    5. Press both steering levers forward.
    6. Once driver receives a velocity update, schedule an update for the time when the tank should be at the next route point.
    7. Go back to item one.

- **Drive Shaft** The drive shaft passes the RPM's from the engine to the transmission. If it is damaged the output will be less than the input.

- **Engine** - The engine provides the power to the driveshaft in the form of revolutions per minute. The output RPM is computed from the fuel flow rate and damage level.

- **Fuel Tank** The fuel tank provides a fuel source for the engine. Fuel stored is assumed to be the correct fuel required (JP-4, diesel, gas, etc). The fuel tank computes either the time when the tank will be full (given an input fuel flow rate greater than the

output rate), or when it will be empty and schedules an update for that time. When an update arrives the status of the tank is recomputed and if the tank is empty the fuel flow rate is set to zero, causing the engine to quit. If the tank is full and fuel is still being input at a greater rate than it is being used a *kspillFuel* event is issued with the excess flow rate as the value.

- **Gear Shifter** The gear shifter provides the control point from which the gear is selected. The gears themselves are located in the transmission. The gear is changed with the *kgear* event, which causes the new value to be sent to the transmission.

- **Mini Gun** The mini gun is the gun located on top of the tank, controlled by the commander. It can be moved in azimuth and elevation, reloaded with ammunition, and fired. Each bullet fired passes the starting location and initial velocity vector(x,y,z) to the environment.

- **Powered Wheel** The powered wheel translates the rotational velocity (RPM) into a forward velocity. The distance is determined by subtracting the braking and slippage revolutions from the forward revolutions produced by the transmission and multiplying this factor by the circumference of the wheel. The velocity is then calculated using the distance traveled and the time since the last update.

- **Steering Lever** The steering lever is the control point the driver manipulates to steer the tank. Input control values range from zero to one. A zero input translates to no forward motion in the track, while a one input results in the maximum forward motion with the given RPM from the drivetrain.

- **Throttle** The throttle is the control point the driver manipulates to control the speed of the tank. Input control values for the throttle also range from zero to one. The throttle controls the flow of fuel from the fuel tank to the engine through the *kchangeThrottle* event.
- **Transmission** The transmission relays the RPM from the driveshaft to the treads based on the gear, the clutch, and the damage level of the transmission. The output revolutions are determined by multiplying the input revolutions by the gear ratio.
- **Wheel** The wheels help to support the treads and mimmick the motion of the powered wheel. The rotational velocity of the wheel is set based upon the forward velocity of the wheel and the wheel's radius.

4.4 Sequential Simulation Processing

Pseudo code for the body of the main simulation program is:

```
Initialize simulation;
Loop:
    Event := Get first event from my next event queue;
    Update to the time of Event;
Until(Done Event);
```

4.4.1 Updating the object. Updating the object to time T requires processing all events on the object's queue with time T. This process may result in the generation of more new events and/or messages to be passed up to the parent. More than one message may be passed to the parent during a single update cycle. This normally occurs when the object is sending communication to an object outside its "scope." Passing a message is

accomplished by directly inserting events into the next event queue of the receiving object. If a child component is passing a message to another object through its parent, the parent component must have an event handler for that event in order to pass it to the correct object. The end of the update cycle inserts the first event from the object's queue into the parent's next event queue in accordance with the event processing protocol.

4.4.2 Processing an Event. Processing an event requires determining and executing the sequence of actions for the event. Those sequences of events are predetermined and are stored in the event handler for each event occurring in the object. Actions may include calculations by the object, calculations by a child object, and generation of messages for other objects. Events requiring calculations by a child object are passed to the child by inserting the event in the child's event queue and then calling the update function with the time of the event for that child, as shown in the following sequence:

```
Child Queue := Child Queue + Event;
Update Child to Event time;
```

4.5 Parallel Simulation Processing

The addition of more processors requires minor changes in the simulation. The changes are limited to alterations in the data inherited by the objects, alterations in the parameters in a small number of methods, and the addition of a few new methods.

4.5.1 Changes to the Object Data. Three new fields have been added:

- **The node of the parent** - This field contains a logical pointer to the parent object.

- **MPI Message Datatype** - This is the simulation message datatype required for MPI. It contains the definition of what fields are included in an event message and how much space each requires.
- **ID** - The id of the object is required to maintain the hierarchical algorithm. It is read from the initialization file for the instance and is used by the parent to correlate the event with the child which produced it. This corresponds to the pointer that existed in the sequential version of the program.

4.5.2 Additional Message Type. Distribution of control and the possibility of more than one event being sent from a child to a parent requires the addition of a *roundDone* message, denoting the last message for the round. This message is the equivalent of a function return.

4.5.3 Changes to the Object Methods. Several methods must be changed to be used in a parallel environment. The *Create*() function requires two parameters: the logical pointer to the parent and the MPI_Datatype definition. The method used to pass a message to parent's event queue requires extra logic to determine if it should insert an event in the queue or send a message. Likewise, any event passed to a child must first go through the same type of checks. This version of RADSIM does not check the parent to child communication because the partitions are hard coded. Either the method to insert an element in a child queue or send a message to the child is inserted directly into the code at the appropriate time.

4.5.4 Additional Methods. New methods are required to send and receive messages. A layer of abstraction is added to the MPI interface by providing utility functions to send and receive simulation events. These functions use non-blocking communication and return the MPI datatype required to wait for communication to finish. New methods are required for partitioning as well. These changes are described in Section 4.5.6.

4.5.5 Message Passing Interface Requirements. The parallelization of the simulation requires the introduction of the *Message Passing Interface*, or MPI. MPI requires a single program to be replicated on each of the processors in the simulation. The main program must then be used to partition the simulation. Pseudo code for initializing and partitioning a simulation is shown below:

```
Initialize MPI;
Initialize Simulation;
myid=GetProcessorNumber();

if(myid=0) then
  doPartition(0);
else if (myid=1) then
  doPartition(1);
....
```

4.5.5.1 Initializing MPI. Every processor must initialize MPI, determine the number of processors used by the program, determine its processor ID, and set up the simulation data type for use with MPI. The first three requirements are satisfied by functions included in the MPI library. Making the simulation event message type a legal MPI datatype is completed in the *MPISetup* function included in the file *basicTypes.c*.

4.5.5.2 Portable Makefile for the Simulation. MPI uses a file entitled *Makefile.in* to increase portability. Instead of creating a makefile, the user can create a *Makefile.in* following the examples given in the MPI documentation. Once this file has been created, moving the program to a new platform simply entails running the *mpireconfig* command to produce the makefile. The example below will reconfigure the makefile for RADSIM if the system has an implementation of MPI installed. This command is then followed by *make* to build RADSIM. The Makefile.in for RADSIM is located in Appendix E.

```
mpireconfig Makefile; make
```

4.5.6 Partitioning. Every partition is assigned to exactly one processor and every processor contains exactly one partition. Each *doPartition(x)* consists of initializing the components in its partition and performing the loop: get event; update to the time of the event. Instead of inserting events in a parent NEQ, partition parent objects send messages to the processor containing the parent object.

4.5.7 Adding Events to Children's NEQ. Objects whose children are located on another processor need to send a message to the processor and receive all the messages back from the child processor and add them to the next event queue of the top level object. Since no *return* statement is executed in the parallel mode, a new message type, *roundDone*, was added to serve as one.

4.5.8 Sending Messages. Adding events to the queue of a remote component requires sending a message. Messages are sent using a non-blocking send. The sending

function returns a request handle so that the user can do more work while waiting for the communication to finish.

4.5.9 Receiving Messages. All components have the capability to receive messages from their parents and children. An object can pass a single event down to a child and process the other required sends for the same event before blocking to receive messages from the child. The child, on the other hand, receives a single event from the parent and updates to that time, sending all events up to the parent and down to its children as needed. When no more events can be processed at the child, it sends the top event in its queue followed by the *roundDone* message.

4.5.10 Updating the Child Partition. Updating the child partition is done automatically. Following the protocol, any message passed to a partition from a parent partition is an admission that no other events with a smaller time will arrive from the parent partition and updating to the time of the event can occur.

4.6 Simulation Verification

The simulation output for the partition configurations chosen is verified by two methods. The first method uses the position output from the tank and the simulation time for the sequential configuration along with each of the new configurations. The output is examined for inconsistencies in output, identifying a parallel model which is not correctly modeling the system. The second method used for verification was to examine the number and types of events which occurred during the simulation. Once again, any differences are the result of a parallel simulation error.

4.7 Summary

This chapter described the implementation of **RADSIM**, the hierarchical simulation built for this research effort. The hierarchical decomposition of the main battle tank simulation player was given. The implementation details were given for the sequential algorithm. The changes needed to transform the sequential version into the parallel version were described. Finally the verification methods were described. Information on obtaining and using RADSIM is located in Appendix D.

V. Results and Analysis

5.1 Introduction

This chapter describes the generation of three simplistic analytic models for performance evaluation of different partition configurations. A description is given for the test cases generated to validate the analytic models. Results and analyses of the analytic model validation are given.

5.2 Analytic Model

5.2.1 Analytic Model Background. *The intent of this analytic model is to provide a means to sort a set of simulation partition configurations based on runtime.* The technique used to develop the analytic model is to start with a simplistic model and incrementally refine the model as required to accurately choose the partition with shortest runtime from a set of two partitions. Probabilistic modeling was chosen as the general modeling paradigm to construct the analytic model.

5.2.2 Notation. Figures 15 and 16 are used to explain the following notations and definitions:

- $EventKinds$ = The set of different types of events
- P_i = Probability of event i occurring, where $i \in EventKinds$
- R_i = Runtime of event i. That is, if event i is injected into the system at level x of the hierarchy, the event runtime is how long it takes to return to level x. R_i is either

T_{seq_x} or T_{p_x} corresponding to a single processor implementation or a multiprocessor implementation respectively.

- E_{total} = The total number of events in a simulation run.
- T_{saved_x} = The time saved (relative to the sequential program) for event x.
- **Partition** - The set of object instances that run on the same processor. For instance from Figure 15: Partition P of Configuration A (denoted P_P^A) contains the *Main Battle Tank, Body, Driver,* and *Mini Gun* objects.

$$P_P^A = \{Main\ Battle\ Tank, Body, Driver, Mini\ Gun\} \quad (1)$$

$$P_Q^A = \{Commander\} \quad (2)$$

$$P_P^B = \{Main\ Battle\ Tank, Body, Driver\} \quad (3)$$

$$P_Q^B = \{Mini\ Gun\} \quad (4)$$

$$P_R^B = \{Commander\} \quad (5)$$

- **Partition Configuration** (PC) - A set of partitions that covers the objects in the simulation for a particular run. Examples include Configuration A and Configuration B in Figure 15, denoted PC_A and PC_B respectively. For example $PC_A = \{P_P^A,\ P_Q^A\}$.
- **Partition Boundary Object** - An object, o, is a partition boundary object if either its parent or its child is a member of another partition. The set of partition boundary objects for Configuration A are denoted by $PBOs_A$. Therefore, the partition boundary objects for the two partition configurations shown in Figure 15 are as

follows:

$$PBOs_A = \{Main\ Battle\ Tank, Commander\} \quad (6)$$

$$PBOs_B = \{Main\ Battle\ Tank, Mini\ Gun, Commander\} \quad (7)$$

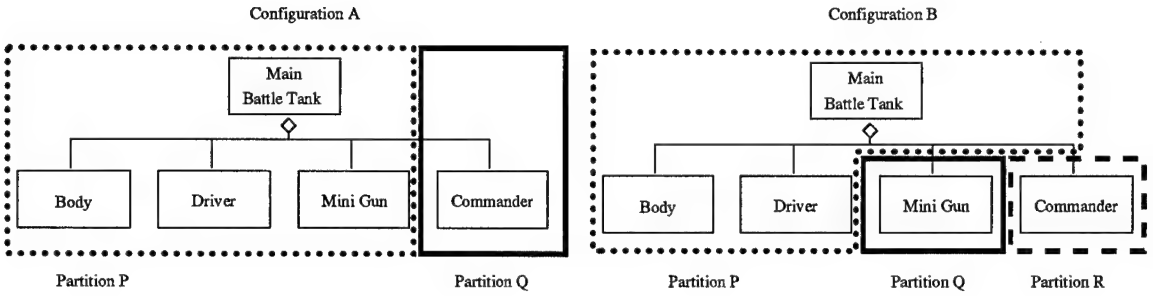

Figure 15. Two Possible Partition Configurations

- **Parent Partition Boundary Object (PPBO)** - A PBO is a PPBO relative to another PBO if the other PBO is in another partition and is on a lower level of the hierarchy.

- **Child Partition Boundary Object (CPBO)** - A PBO is a CPBO relative to another PBO if the other PBO is in another partition and is on a higher level of the hierarchy.

- **Event Parallelism** - Event Parallelism is the number of simultaneously activated partitions due to event x. The event parallelism due to event x for PC_A is denoted: $\|_x^A$. Therefore, given that the RADSIM code sends the *xpositionUpdate* event to the *Commander*, *Mini Gun*, and *Driver*, the resulting event parallelism for *xpositionUpdate* is

$$\|^{A}_{xpositionUpdate} = 2 \tag{8}$$

$$\|^{B}_{xpositionUpdate} = 3 \tag{9}$$

$$\forall x \in EventKinds, \forall Z \in Partition\ Configurations \mid 1 \leq \|^{Z}_{x} \leq total\ partitions \tag{10}$$

- **Event x Parallel Runtime** - T_{p_x} (See Figure 16) T_{p_x} is the amount of time needed to run the simulation given some multiprocessing partition configuration.
- **Event x Sequential Runtime** - T_{seq_x} (See Figure 16) T_{seq_x} is the time needed to run the event on a sequential machine.
- **Event x Overhead** - OH^1_x (See Figure 16) The overhead from both sending and receiving messages. OH^1_x is read "the overhead due to sending and/or receiving event x from child 1."

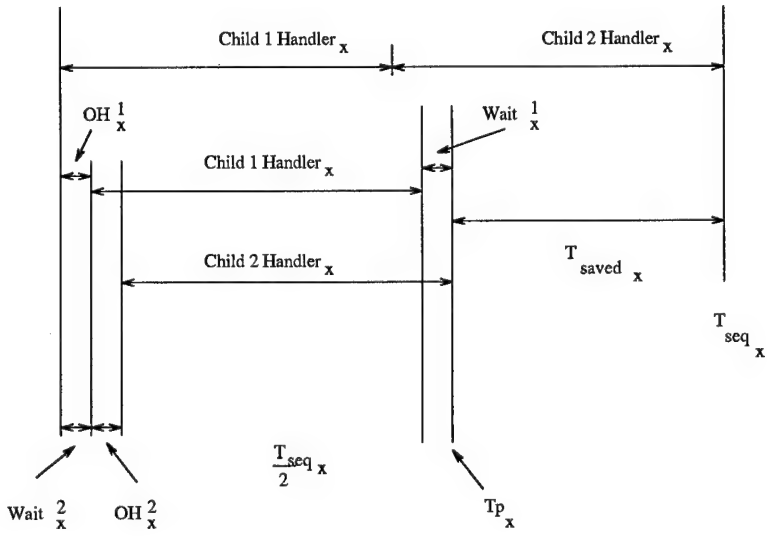

Figure 16. Event X Runtime Timing Diagram for an Object O

5.2.3 Simplistic Model. For a particular partition configuration(z), given a set of events, $\forall i \in EventKinds$, if P_i is known, the runtime contribution of each event, R_i is known, and the total number of events, E_{total} is known, the runtime can be calculated:

$$PC_z\ Runtime = E_{total} \sum_{\forall i \in EventKinds} R_i P_i \tag{11}$$

This model requires previous knowledge of the event runtimes, the probability of an occurrence of an event, and the total number of events. This information may be known in certain cases. For instance, if a particular simulation were run to time 100 seconds and statistics collected, the runtime of the same run extended to 200 seconds could be determined assuming events occurred with the same probabilities in both the first and second 100 seconds. Validation of this model is done in Section 5.3.1.

5.2.4 Simplistic Refinement One. The first refinement to the simplistic model is to remove the total number of events occurring in the simulation. The resulting equation (Equation 12) can no longer predict the runtime of the simulation. However, it can be used to predict the average runtime per event for any partition configuration. Therefore, this equation can still be used to evaluate one partition configuration vs another, and thus is able to order a set of partition configurations. Validation of this model is done in Section 5.3.2.

$$PC_z\ \frac{Runtime}{event} = \sum_{\forall i \in EventKinds} R_i P_i \tag{12}$$

As is clearly evident, this model is not much better than the first. It still requires the runtimes and probabilities for all events. The next refinement decomposes R_i in an attempt to eliminate many of the event runtime calculations.

5.2.5 Refinement Two. Recall that R_i is the runtime for a given event type i. Given a particular simulation and two partition configurations, the probabilities of events occurring is exactly the same. The only differences are the number of partition boundary crossings for any particular event and the amount of parallelism inherent in the single event at the partition boundary.

5.2.5.1 Comparing Two Partitions. A simplistic rating is developed to compare two partition configurations. The simplistic rating corresponds to the time saved by using the given partition configuration and parallel processors verses the sequential version of the program. Therefore a negative number signifies a partition configuration which takes longer to run than the sequential version of the program. The calculation need only be made for events triggered or handled by PBO's. This corresponds to only those events which cross the partition boundaries. If the assumption is made that the event's runtime is equally divided among the children, a particular event's time savings can be determined by taking the sequential runtime of the event, dividing it by the number of partitions performing the work, and subtracting the overhead required to send the event to the child object. Passing any information from the child to the parent across the partition is assumed to be negligible because the parent can be doing other work. The overhead for

all events in all children is assumed to be constant, therefore overhead is denoted OH.

$$T_{p_x} = \frac{T_{seq_x}}{\|_x} + OH\|_x \tag{13}$$

$$T_{saved_x} = T_{seq_x} - T_{p_x} \tag{14}$$

$$= T_{seq_x} - \left(\frac{T_{seq_x}}{\|_x} + OH\|_x\right) \tag{15}$$

$$= T_{seq_x}\left(1 - \frac{1}{\|_x}\right) - OH\|_x \tag{16}$$

Therefore the determination of a simplistic rating is done as follows:

$$simplistic\ rating = \sum_{\forall i,\ events\ crossing\ boundary} T_{saved_i} P_i \tag{17}$$

$$= \sum_{\forall i,\ events\ crossing\ boundary} P_i \left[T_{seq_i}\left(1 - \frac{1}{\|_i}\right) - OH\|_i\right] \tag{18}$$

Recall that $\|_x$ is static and is based on only the event handler for event x (located in the parent of the PBO). This successfully replaces a trial runtime of an event for a particular partition configuration with a calculation based on the sequential time of the event, T_{seq_x}, the overhead for any event, OH, and the inherent parallel nature of the event, $\|_x$. Validation is shown in Section 5.3.3.

5.3 Experiment Design

Three configurations were designed to begin the process of model validation. Three different parallel and distributed computers, two possible routes, and two values for the number of spin loops provide many different run combinations. These parameters are described below. Metrics corresponding to OH, P_i, and R_i (T_{seq_i} and T_{p_i}) were collected as needed for the different analytical models.

- **Partitions** - The three partition configurations chosen for the experiments were a sequential version, a two processor version, and a three processor version. The two and three processor partition configurations are shown in Figure 17 and Figure 18 respectively. The partition configurations listed in Table 6 are labeled as follows: *PC* 1 is the sequential version of the program, *PC* 2 corresponds to the two processor configuration shown in Figure 17, and *PC* 3 corresponds to the three processor configuration shown in Figure 18.

- **Routes** - Two routes were chosen to validate the models. The first route has only 52 route points, while second contains 97 route points.

- **Spin Loops** - There are two different values for spin loops: 1000 and 10000. Each iteration through the spin loop executes a floating point divide on a double.

- **Machines** - The simulations were run on three machines: The Paragon XP/S, the SGI Power Challenge, and a network of Sun SparcStation 20s.

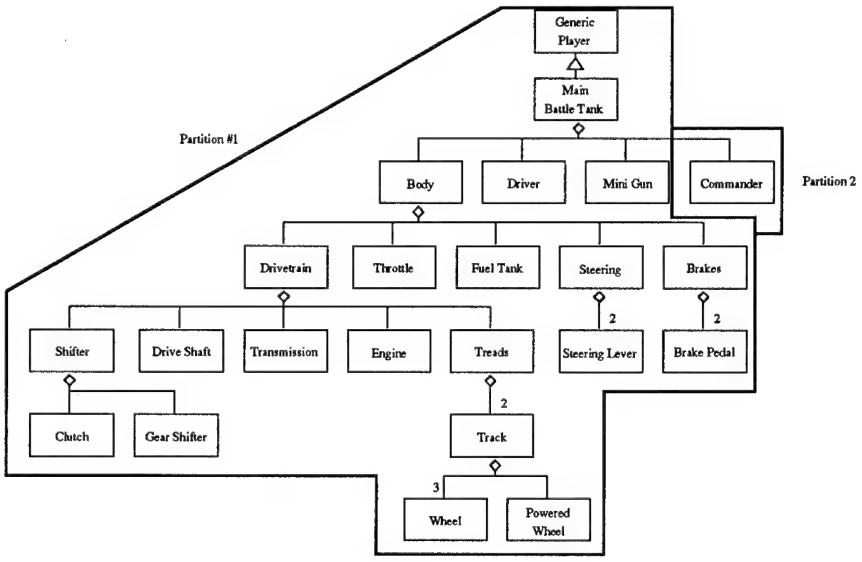

Figure 17. Experiment Two Processor Partition Configuration

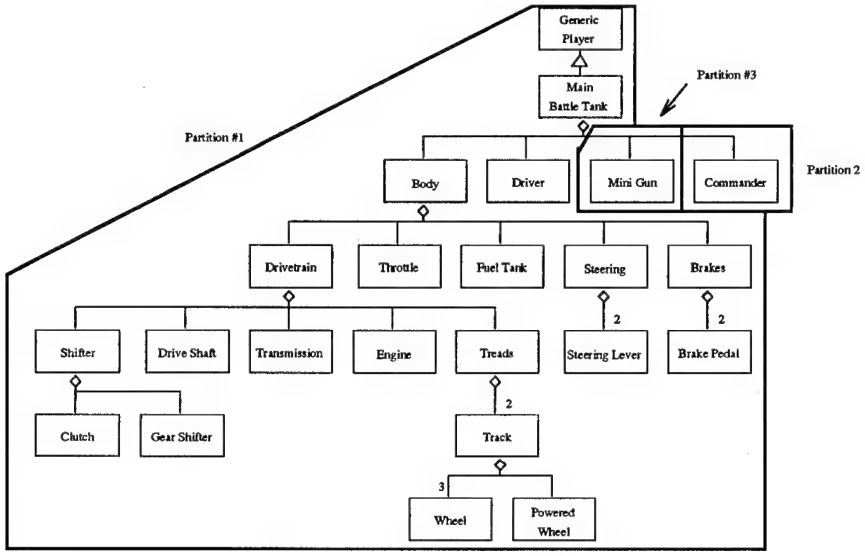

Figure 18. Experiment Three Processor Partition Configuration

5.3.1 Validation of the Simplistic Analytical Model. According to Equation 11 the simplistic model can be used to determine runtime of a partition given the probability of an event occurring, the runtimes of the events, and the total number of events in the simulation. Since there are no external inputs and the same sequence of events is run for each route point, the simplistic model can be applied. Application of the simplistic model predicts the runtime of the second route based on the statistics generated for the first route. This technique can only be applied on runs made with the same partition and number of spin loops. If these values changed, the runtime of each of the events is changed.

5.3.1.1 Simplistic Model Results and Analysis. The runtimes of the simulation runs are calculated by taking the ratio of the total number of events in a run with unknown runtime to the total number of events in a run with a known runtime and multiplying it by the known simulation runtime for the partition: $Runtime_{run\ 2} = \frac{E_{total\ run\ 2}}{E_{total\ run\ 1}} Runtime_{run\ 1}$ The results of performing this calculation are shown in the *Sim-*

plistic Model column of Table 6. There are 1964 events in the simulation using the short route and 5152 events in the simulation for the longer route.

Speedup is calculated based on the definition from Kumar(23) shown in Equation 23. The values for the speedup are shown in Table 6. The definition of percent change is given in the following equations:

$$SM = Simplistic\ Model\ time \tag{19}$$

$$\%Ch = Percent\ Change \tag{20}$$

$$time = Actual\ runtime \tag{21}$$

$$\%Ch = \left(\frac{SM - time}{time}\right) 100.0 \tag{22}$$

$$Speedup = \frac{Best\ Sequential\ Runtime}{Parallel\ Runtime} \tag{23}$$

Analysis of the data collected suggests that the simple model performs better with a higher computation to communication ratio. The simplistic model is always closer on the runs with 10k spin loops than on those with only 1k spin loops. Speedup values also show the importance of a larger computation to communication ratio. The higher this ratio, the closer the speedup will be to the maximum for the partition configuration. The speedup should remain close to constant for both routes given the same partition and the same machine. In fact, this holds true for all partitions.

There are also some unexpected results. One might expect that taking a longer run and predicting a shorter run's time would produce a more accurate result, where accuracy is defined in terms of the percent change from the actual to the predicted runtime. The SGI and Paragon runs were more accurate when predicting the runtime of a shorter run, while the network of SparcStations produced the opposite result.

Also, it is interesting that the best overall runtime was on the network of SparcStations, rather than on either of the parallel machines. However, this fact can be explained by recalling the fact that each iteration of a spin loop executes a floating point divide on a double. The SGI and Paragon machines are executing this divide in software rather than in hardware.

Since the compilers on the machines are not the same, it is incorrect to make any further comparisons of overall runtime. However, speedup values should remove the compiler differences. If this fact is true, one would expect the speedup value on the SGI to provide the best results since it is a shared memory machine, and hence, communication between processes should be shorter. The SGI does in fact produce the best speedup results.

5.3.2 Validation of Refinement One of the Simplistic Analytical Model. Validation of the first refinement of the simplistic model was conducted with the same experiment test cases as the simplistic model.

5.3.2.1 Refinement One of the Simplistic Model Results and Analysis. Refinement one allows the ordering of the partition combinations by summing the runtimes per event multiplied by the probability of occurrence of that event. The results of

applying the refined simplistic model to the three partition configurations are shown in Equations 24, 25, and 26.

This being the case the partition configurations are ordered:

$$SGI\ Power\ Challenge = 2,\ 3,\ 1 \tag{24}$$

$$Network\ of\ Suns = 2,\ 3,\ 1 \tag{25}$$

$$Paragon\ XP/S = 3,\ 2,\ 1 \tag{26}$$

The ordering is what was expected except for the Paragon runs. Configuration three was consistently better for all combinations of route and spin loops on the Paragon. More test cases and metrics need to be generated to explain this phenomenon. All indications during the run would suggest that the order of partition configurations would be aligned with both the Sun and the SGI runs. The command *showpart* showed the three partitions of configuration three being assigned to a triangle of processors. This triangle configuration forces communication to travel along both horizontal and vertical mesh elements, which has been shown to have longer delays than sending only one of the two mesh elements(33). Furthermore, no other users were using the interactive partition, limiting the amount of *cut-through* communication.

5.3.3 Validation of Refinement Two. Validation of refinement two also uses the same experiment test cases as the other analytical models. However, the analytical model requires the computation of several new parameters from the RADSIM code. These pa-

rameters include the event parallelism for all events in the partition boundary objects, the sequential time of the events, and the overhead for any event, OH. The event parallelism remains constant for all machines while the other parameters do not. Table 7 shows the event parallelism in the *Main Battle Tank Object* for the partition configurations chosen (only those events which cross partition boundaries).

The overhead for sending a message was determined by recording the time to send 10k messages and dividing by 10k. The results for the three machines are listed in Table 8. Table 10 shows the probability of each of the parent partition boundary object's events occurring. Finally, the sequential runtimes for the events given the different cases for number of spin loops and machine are shown below in Table 9.

5.3.3.1 Refinement Two Results and Analysis. Refinement two allows the ordering of the partition configuration based on the sequential time of the event, T_{seq_e}, the overhead for any event, OH, and the inherent parallel nature of the event, $\|_e$, as shown in Equation 27.

$$simplistic\ rating = \sum_{\forall i,\ events\ crossing\ boundary} P_e \left[T_{seq_e} \left(1 - \frac{1}{\|_e} \right) - OH\|_e \right] \qquad (27)$$

The PBO for configuration one (the sequential version of the program) is $\emptyset$, therefore the simplistic rating is 0. Tables 11 and 12 show the results of applying Equation 27 to configurations two and three, respectively.

The order of partition configurations produced by refinement two is as follows:

- Suns, running 1k spin loops: 1, 2, 3.
- Suns, running 10k spin loops: 3, 2, 1.
- SGI, running 1k spin loops: 1, 2, 3.
- SGI, running 10k spin loops: 3, 2, 1.
- Paragon, running 1k spin loops: 3, 2, 1.
- Paragon, running 10k spin loops: 3, 2, 1.

This model accurately orders the the partition configurations when the objects execute 1000 spin loops on each event. However, the model inaccurately orders the partition configurations when the objects execute 10000 spin loops. Inaccuracies are largely due to the assumptions made in the model and timing inaccuracies. The model does not account for multiple communications from the child to the parent during the run, only from parent to child. Also, the model does not include the updates the child schedules for itself with the parent object.

5.4 Summary

This chapter introduced three simplistic analytic models for determining if one partition configuration was better than another for certain given conditions. The models were used to predict the order of runtimes for three test partition configurations with some success. The simplistic model and the direct derivation from the simplistic model accurately predicted the ordering of the configurations, but require runtime and probability of occurrence information for each event in the simulation. The refinement to the simplistic models replaced the necessity to know the runtime of each event with static conditions of the ob-

jects in the simulation. This model correctly ordered the partition configurations when the number of spin loops executed by each event was low, but ordered them incorrectly when the number of spin loops was high.

Table 6. Runtimes

Machine SGI/Sun	Spin Loops 1K/10K	Route (1,2)	PC (1,2,3)	time (sec)	Speedup	Simplistic Model	Percent Change
Sun	1K	1	1	1.070		0.939	-12.24
			2	3.732	0.287	2.959	-20.71
			3	6.862	0.156	4.081	-40.52
		2	1	2.463		2.807	13.97
			2	7.763	0.317	9.790	26.11
			3	10.705	0.230	18.000	68.15
	10K	1	1	79.542		77.863	-2.11
			2	61.149	1.301	57.375	-6.17
			3	62.539	1.272	60.262	-3.64
		2	1	204.251		208.656	2.16
			2	150.508	1.357	160.407	6.58
			3	158.080	1.292	164.053	3.78
SGI	1K	1	1	5.708		4.4308	-22.37
			2	6.832	0.835	5.1193	-25.07
			3	11.623	0.491	11.357	-2.28
		2	1	13.429		14.9733	11.49
			2	16.211	0.828	17.9218	10.55
			3	29.792	0.451	30.4897	2.34
	10K	1	1	924.426		887.7631	-3.97
			2	665.602	1.389	641.0318	-3.69
			3	686.797	1.346	663.6128	-3.37
		2	1	2328.796		2425.0	4.12
			2	1681.566	1.385	1746.0	3.83
			3	1740.801	1.338	1801.6	3.49
Paragon	1K	1	1	36.641		34.5995	-5.57
			2	32.088	1.142	30.1919	-5.91
			3	30.308	1.209	28.5783	-5.71
		2	1	90.762		96.1173	5.90
			2	79.200	1.146	84.1738	6.27
			3	74.967	1.211	79.5045	6.04
	10K	1	1	7302.267		7027.8	-3.77
		2	1	18435.371		19155.0	3.90

Table 7. Event Parallelism for the Main Battle Tank Object

Event	Event Parallelism	
	PC_2	PC_3
X position updt	2	3
Y position updt	2	3
Z position updt	2	3
heading updt	2	3
velocity updt	2	3
rotational velocity updt	2	3
pull trigger	-	1
move gun in El	-	1

Table 8. Message Sending Overhead

Machine	Overhead
Network of Suns	0.000866
SGI Power Challenge	0.003966
Paragon XP/S	0.000072

Table 9. Sequential Runtimes for Events

Event	Sun		SGI		Paragon	
	1k ($x10^{-3}$)	10k	1k	10k	1k	10k
X position updt	0.4032	0.0877	0.0023	1.019	0.016	8.070
Y position updt	0.3194	0.0861	0.0021	1.019	0.016	8.045
Z position updt	0.0539	0.0004	0.0006	0.005	0.004	0.039
heading updt	0.3419	0.0867	0.0022	1.019	0.016	8.051
velocity updt	0.3176	0.0871	0.0021	1.019	0.016	8.051
rotational velocity updt	0.3237	0.0869	0.0022	1.019	0.016	8.048
pull trigger	0.3161	0.0842	0.0018	0.971	0.012	7.655
move gun in El	0.1329	0.0408	0.0010	0.479	0.008	3.780

Table 10. Event Probabilities

Event	Probability
X position updt	0.0545
Y position updt	0.0545
Z position updt	0.0545
heading updt	0.0545
velocity updt	0.0545
rotational velocity updt	0.0545
pull trigger	0.0275
move gun in El	0.0270

Table 11. Simplistic Rating for PC_2

Event	Sun		SGI		Paragon	
	1k	10k	1k	10k	1k	10k
X position updt	-0.00008	0.0023	-0.0004	0.0298	0.0004	0.2194
Y position updt	-0.00009	0.0023	-0.0004	0.0298	0.0004	0.2194
Z position updt	-0.00009	-0.00008	-0.0004	-0.0002	0.0001	0.001
heading updt	-0.00009	0.0023	-0.0004	0.0298	0.0004	0.2194
velocity updt	-0.00009	0.0023	-0.0004	0.0298	0.0004	0.2194
rotational velocity updt	-0.00009	0.0023	-0.0004	0.0298	0.0004	0.2194
Sum	-0.00053	0.0114	-0.0024	0.1486	0.0004	1.098

Table 12. Simplistic Rating for PC_3

Event	Sun		SGI		Paragon	
	1k	10k	1k	10k	1k	10k
X position updt	-0.0001	0.0030	-0.0006	0.0364	0.0006	0.294
Y position updt	-0.0001	0.0031	-0.0006	0.0364	0.0006	0.294
Z position updt	-0.0001	-0.0001	-0.0006	-0.0006	0.0001	0.001
heading updt	-0.0001	0.0030	-0.0006	0.0364	0.0006	0.29
velocity updt	-0.0001	0.0030	-0.0006	0.0364	0.0006	0.294
rotational velocity updt	-0.0001	0.0030	-0.0006	0.0364	0.0006	0.294
pull trigger	-0.0002	-0.0002	-0.0001	-0.0001	-0	-0
move gun in El	-0.0002	-0.0002	-0.0001	-0.0001	-0	-0
Sum	-0.0010	0.0148	-0.0038	0.1812	0.0031	1.47

VI. Conclusions and Recommendations

6.1 Introduction

This chapter has three main objectives. Analyze the results with respect to the objectives and goals of the research, describe the contributions to the simulation community and the military, and finally, to provide recommendations for future study of parallel hierarchical simulation.

The objectives of this research effort were to develop both a hierarchical sequential discrete event battlefield simulation and a hierarchical parallel discrete event simulation, and to characterize the runtime of the simulations. The goal was to identify the major factors in determining runtime of the simulation. The objectives were met. Both sequential and parallel versions of RADSIM were constructed to simulate a main battle tank. Three simplistic analytic models were developed to determine which of two partition configurations would produce the lowest runtime.

Analysis of the analytic models and results of the test cases showed that runtime can be modeled to the degree necessary to correctly choose one partition configuration over another under certain conditions and that partition objects based on aggregate components can produce speedup. The simplistic models derived predicted the lower-time partition configuration in most of the cases. However, the most accurate simplistic models require an extensive knowledge of the runtime of each event, and the probability that each event will occur. The third analytic model replaces the need for the specific runtime of each event in the partition with a calculation based on the sequential runtime of the event, the inherent parallelism of the event, and the overhead of sending an event. In its current form

this model is not as accurate as the other models. However, with more refinements it may be possible for this model to produce accurate assessments of the partition configurations in much less time. The factors contributing to the runtime of the simulation are the event parallelism, the amount of work required by each event, the overhead of sending an event to a remote processor, and the probability that the event will occur.

The theoretical speedup of the simulation is limited by the number of events causing simultaneous action in more than one partition and their probability of occurrence. In actuality, the speedup is also limited by the conservative nature of the synchronization control structure. The current control structure requires each processor to spend a large amount of time waiting for the object to get an event from its parent. This time could be better spent performing some calculation.

6.2 Research Contribution

This research effort contributed to current general knowledge in two ways. It provided a base hierarchical simulation model, which future students can use for further exploration. It has an accurate motion model for a tank given two tread speeds, so it could be used to experiment with Artificial Intelligence controllers. Secondly, the research has provided a method to sort partition configurations based on runtime. This could be incorporated into a method of dynamically partitioning the simulation. Processors with free time could compare the current partition configuration with other possible partition configurations. If another partition configuration was found that was significantly better than the current one, the objects could be moved to the new configuration.

6.3 Recommendations for Further Study

The research completed was a good base step. However, many improvements could be made to enhance the concurrency of the algorithm. Hierarchical algorithms are good at exploiting concurrency due to a single event, such as the position updates in the tank simulation. On the other hand, spatial partitioning is good at determining proximity alarms and collision detection. Future research should look at merging these two methods of partitioning a simulation, absorbing the better qualities of each. It is entirely possible that providing both partitioning methods could lead to further runtime reductions.

Future students could test the capability of a hybrid synchronization protocol to reduce runtime. The conservative protocol is restricting in that all processors must wait until they receive either an update or an event from the parent. Metrics collected showed that most of the events executed are simply updates to a child. Since this is the case, all processors should save the current state and proceed executing the top event in the queue. If an event does arrive in the past, the processor could pop the previous state and execute the next event.

The partition configuration is currently hard coded. Future research could finish the automatic partition configuration to handle greater numbers of processors. Some of the objects currently contain the code to implement the configuration automatically. This code could be replicated in the remaining object's code.

Currently the messages passed between objects contain only a single event. This could be changed to allow multiple events to be passed in a single message. The parallel version would derive the most benefit, since the cost of communication is high in relation

to the number of calculations that can be performed. At the least this would eliminate the need for a communication containing the *roundDone* message directly following another message from the same object.

The simplistic analytic models should be developed further to provide the means to quickly evaluate two partition configurations, particularly, the *Refinement Two* model. This model successfully removes the runtime of an event on a particular partition configuration, but it still contains the probability of each event occurring. Improvements should be made to the accuracy of the predictions and every possible effort should be made to eliminate the probability of an event occurring from the model equation.

6.4 Summary

Simulations continue to grow in both size and fidelity. These increases require a large amount of computational resources to complete the simulations within the required time. Parallel computers provide the required resources, but produce the added requirements of dividing the simulation to run on multiple processors and synchronizing the processors in order to retain the correct time-ordering of events.

This thesis investigated the effects of a hierarchical partition and tree of aggregate components as one combination of a partitioning algorithm and method of synchronization. A simulation of a main battle tank, several test cases, and three analytical models were constructed to determine if this combination of partitioning method and synchronization control could produce speedup and to characterize the runtime of one partition configuration verses another.

The simulation produced a speedup of approximately 1.4 for a two processor case running on the Silicon Graphics Power Challenge. The analytic models were able to choose the partition configuration from two choices in most instances. The factors which contributed to the runtime of any particular partition configuration were determined to be an event's parallelism, the amount of work required for the event, the overhead of sending the event to another partition, and the probability of the event occurring.

Appendix A. Definitions

- **Discrete Event Simulation** - A simulation in which time advances based on when the next event occurs.
- **Hierarchical Object** - An object composed of smaller pieces. For example, one hierarchical breakdown of a wheel is into a tire and a rim.
- **Component** - A piece of a hierarchical breakdown. Components can be elements or assemblies.
- **Element** - An element is the lowest level piece in the hierarchy. It is broken down to its lowest level.
- **Assembly** - An assembly is any component that is further broken down into other assemblies or elements.
- **Player** - A top level component that interacts with the simulation.
- **Sequential Computing** - Computing performed on a single processor.
- **Parallel Computing** - Computing performed on two or more processors.
- **Discrete Event Simulation (DES)** - State of the simulation is determined by the event with the lowest time. Simulations generally include a next event queue and clock as well as the normal simulation players. DESs have the ability to advance time in unequal increments.
- **Time-Driven Simulations** - Simulation time progresses based on a fixed time increment.

- **Object-Oriented Simulation** - Simulations are built using object-oriented means: with data and the functions to modify the data included in one package.
- **Causality** - Occurring in the correct time-order. If one event occurs before another and affects it, it must be executed first in order to keep the simulation in the correct state.
- **Synchronization** - Methods used to keep the simulation in correct time order on multiprocessing platforms. Generally includes both optimistic (16) and conservative protocols (10).
- **Conservative** - Conservative synchronization protocols maintain correct time ordering at all times. The simulation proceeds when all input channels have an event, and it proceeds with the lowest event time.
- **Optimistic** - The simulation is allowed to continue even though it may not be in correct time order. State is saved periodically. If a causality error occurs, time is rolled back to a time when the state was correct.
- **Partitioning** - Dividing the data and functions of a program in order to put it on a distributed or parallel computer.
- **Logical Process (LP)** - A software process running on the node of a parallel computer.
- **Message Passing Library (MPL)** - The native communications library for the IBM SP2.
- **Message-Passing Interface (MPI)** - A freely available message passing library for most parallel machines.

- **Parallel Virtual Machine (PVM)** - A freely available library for running parallel programs on computers in a network, or parallel processing machines.
- **PRAM** - Parallel Random Access Machine. A computational model in which all processors have access to all memory simultaneously, if needed.

Appendix B. RADGEN User's Guide

B.1 Using the Program

Use of the RADGEN program is very easy. Create a object description file following the rules given below (object.dsc) and run RADGEN with that file as the parameter:

```
radgen object.dsc
```

This action will automatically create the files object.c and object.h

B.2 Description File Contents

B.2.1 Component Kind and Filename. The component kind is the first string in the file. It consists of *.element*, *.assembly*, or *.player* signifying an element, assembly, or player respectively. The second string is the name that will be given to the file (X.c and X.h) and the datatype. An example:

```
.element  miniGun
```

B.2.2 Method Prefix. The prefix prepended to all methods of the object is designated by the line:

```
.smallname mGn
```

B.2.3 Standard Includes. Standard includes (i.e. *#include <xx.h>*) are designated in a group. The keyword *.stdincludes* begins the group and the keyword *.endstdincludes* designates the end of the group. All lines between the beginning designator and the ending designator are read, and it is added to the *Filename.h* file in the form *#include*

<*xxx.h*> , where *xxx* is the line read. Standard include files for the IO (stdio.h), the standard library (stdlib.h), and the strings (strings.h) are automatically added and need not be put into the group. An example group:

```
.stdincludes
test1
test2
.endstdincludes
```

B.2.4 Quoted Includes. Quoted includes (i.e. *#include "xx.h"*) are designated in a group. The keyword *.qincludes* begins the group and the keyword *.endqincludes* ends the group. All lines between the beginning designator and the ending designator are read, and it is added to the *Filename.h* file in the form *#include "xxx.h"* , where *xxx* is the line read. Quoted include files for the next event queue (neq.h), basicTypes (basicTypes.h), and eventTypes (eventTypes.h) are automatically added and need not be put into the group. An example group:

```
.qincludes
barrel
ammoStore
ammoInFlt
.endqincludes
```

B.2.5 Set and Get Attributes. Data items in the object requiring both a *Set* and *Get* method are designated in a group. The keyword *.sgattributes* begins the group and the keyword *.endsgattributes* ends the group. All lines between the beginning designator and the ending designator should be in the form: *kind name*, where *kind* is the C type of the variable (e.g. float, double, int, etc) and name is the name of the variable to be used in

the object. The attribute is added to the object's data elements and *Get* and *Set* methods are generated. An example group:

```
.sgattributes
int triggerState
.endsgattributes
```

B.2.6 File Attributes. Data items in the object requiring to be read initially from a file are designated in a group. The keyword *.fattributes* begins the group and the keyword *.endsgattributes* ends the group. The first line in the group is the file name that should be used to read the data. All lines between the file name and the ending designator should be in the form: *kind name*, where *kind* is the C type of the variable (e.g. float, double, int, etc) and name is the name of the variable to be used in the object. The attribute is added to the object's data elements, a *Get* method is generated, and a line is generated in the *objInit()* function to read the data item from a file. An example group:

```
.fattributes
auto_gun.init  /* file name */
float  firingRate
.endfattributes
```

B.2.7 Initializers. Data items in the object requiring to be initialized are designated in a group. The keyword *.init* begins the group and the keyword *.endinit* ends the group. All lines within the group should be in the form: *name value*, where *name* is the name of the variable and value is the value it should be assigned at initialization. An example group:

```
.init
triggerState 0
.endinit
```

B.2.8 Children. Children of the object are designated in a group. The keyword *.children* begins the group and the keyword *.endchildren* ends the group. All lines within the group should be in the form: *smallname variablename*, where *smallname* is the name that should be prepended onto all actions using that variable. The variable name is used to create a data item in the object. An example group:

```
.children
 bar theBarrel
.endchildren
```

B.2.9 Events. Events affecting the object are designated in a group. The keyword *.eventsIn* begins the group and the keyword *.endeventsIn* ends the group. All lines within the group should be in the form: *name*, where *name* is the name given to the event-handling function. An example group:

```
.eventsIn
pointAt
pullTrigger
releaseTrigger
.endeventsIn
```

B.2.10 Internal Functions. Internal functions are designated in a group. The group begins with *.iFunction* and ends with *.endiFunction.* Each line in the group causes a function to be built with the name provided. An example group:

```
.iFunction
fire
.endiFunction
```

B.2.11 End of File. The end of the file is designated by a *.end* command.

Appendix C. Example Discription files

C.1 Drivetrain

```
.assembly driveTrain
.smallname dTrn

.qincludes
engine
driveshaft
transmission
shifter
treads
terrain
starter
.endqincludes

.dattributes
ThreeD_real_vector position
ThreeD_real_vector velocity
ThreeD_real_vector acceleration
.enddattributes

.children
eng theEngine
dSft theDriveShaft
tmsn theTransmission
sftr theShifter
trd theTreads
.endchildren

.eventsIn
/* from outside */
start
changeThrottle
changeFFR
pressClutch
releaseClutch
gear

/* driveshaft */
updatedRPM

/* transmission */
notcoast
coast
```

```
newRevs

/* engine */
exceedFFR
updateRPM
.endeventsIn

.end
```

C.2 Driver

```
.assembly driver
.smallname dvr

.stdincludes   /* standard include files */
math
.endstdincludes

.qincludes
aList
.endqincludes

.dattributes
float tiredLevel
.enddattributes

.children
list routeList
.endchildren

.eventsIn
start
xPositionUpdate
yPositionUpdate
velocityUpdate
rotVelocityUpdate
headingUpdate
obstacleWarning
routeUpdate
comm
updatesDone
.endeventsIn

.fattributes /* file-read attributes i.e. constants */
driver.init  /* file name */
float  tiringRate
float  startingTiredLevel
.endfattributes

.iFunction
getPosition
changeCourse
stop
```

```
go
turnLeft
turnRight
.endiFunction

.end
```

C.3 Wheel

```
.element wheel
.smallname whl

.sgattributes
float inRotVelocity
float brakingLevel
.endsgattributes

.fattributes
wheel.init
float radius
.endfattributes

.eventsIn
changeBrakeLevel
changeFwdVelocity
.endeventsIn

.end
```

Appendix D. Obtaining and Using RADSIM

The RADSIM program is the property of the USAF, the Air Force Institute of Technology, and Conrad P. Masshardt. Requests for the program should be directed to Dr. Hartrum at the following address: AFIT/ENG, 2950 P Street, WPAFB OH 45433. The RADSIM simulation can be set up and run using a few commands. Obtain the *radsim.tar* archive and follow the commands listed below to compile and run the program. The MPI library, *mpich*, must be installed prior to running the simulation. The directory with the MPI binary files must be included in the user's path.

```
vulcan% tar xvf *.tar
x makeRadsim, 610 bytes, 2 tape blocks
x radsim.tar.Z, 278504 bytes, 544 tape blocks
vulcan% mkdir theSim
vulcan% makeRadsim theSim
```

Once the program is built running the various versions can be accomplished with the *mpirun* command, as follows (also see the MPI documentation):

```
vulcan% mpirun -np 2 radsim2
```

```
ALL: default
##### User configurable options #####

ARCH        = @ARCH@
COMM        = @COMM@
BOPT        = @BOPT@
P4_DIR      = @P4_DIR@
TOOLS_DIR   = @TOOLS_DIR@
MPIR_HOME   = @MPIR_HOME@
CC          = @CC@
CLINKER     = $(CC)
CCC         = @CPP_COMPILER@
CCLINKER    = $(CCC)
F77         = @F77@
FLINKER     = $(F77)
AR          = @AR@
RANLIB      = @RANLIB@
PROFILING   = $(PMPILIB)
OPTFLAGS    = -O3
#OPTFLAGS   = -O2
#OPTFLAGS   = -g
MPE_GRAPH   = @MPE_GRAPHICS@
#
INCLUDE_DIR = @USER_INCLUDE_PATH@ -I$(MPIR_HOME)/include
DEVICE      = @DEVICE@
DEFINES     = -DTAKE_STATS

### End User configurable options ###
SHELL = /bin/csh

CFLAGS  = @USER_CFLAGS@ $(OPTFLAGS) $(INCLUDE_DIR) -DMPI_$(ARCH) $(MPE_GRAPH)
CCFLAGS = $(CFLAGS)
#FFLAGS = '-qdpc=e'
FFLAGS = $(OPTFLAGS)
MPILIB = $(MPIR_HOME)/lib/$(ARCH)/$(COMM)/libmpi.a
MPIPPLIB = $(MPIR_HOME)/lib/$(ARCH)/$(COMM)/libmpi++.a
LIBS = $(MPILIB) $(LIB_PATH) $(LIB_LIST)
LIB_LIST = -lm
LIBSPP = $(MPIPPLIB) $(LIBS)
# Were not ready to do contrib by default yet.
SUBDIRS = test perftest
TESTDIRS = test
```

```
EXECS =

OBJS = basicTypes.o body.o brakePedal.o brakes.o clutch.o commander.o\
driveTrain.o driver.o driveshaft.o engine.o fuelTank.o gearshifter.o\
miniGun.o neq.o poweredWheel.o shifter.o steering.o tank1.o\
steeringLever.o throttle.o track.o transmission.o treads.o wheel.o\
aList.o

CFILES = basicTypes.c body.c brakePedal.c brakes.c clutch.c commander.c\
driveTrain.c driver.c driveshaft.c engine.c fuelTank.c gearshifter.c\
miniGun.c neq.c poweredWheel.c parradsim.c shifter.c steering.c tank1.c\
steeringLever.c throttle.c track.c transmission.c treads.c wheel.c\
aList.c

HDRS = basicTypes.h body.h brakePedal.h brakes.h clutch.h commander.h\
driveTrain.h driver.h driveshaft.h engine.h fuelTank.h gearshifter.h\
miniGun.h neq.h poweredWheel.h radsim.h shifter.h steering.h tank1.h\
steeringLever.h throttle.h track.h transmission.h treads.h wheel.h\
aList.h


default: radsim

radsim: restar $(MPILIB) radsim.o
cc $(CFLAGS) -o radsim radsim.o libradsim.a $(LIBS)

restar: $(OBJS)
        if(-e libradsim.a) then
          rm libradsim.a
        endif

        ar rcv libradsim.a $(OBJS)
        ranlib libradsim.a

radsim.o: radsim.c Makefile
        cc $(CFLAGS) $(DEFINES) -c radsim.c

neq.o: neq.c neq.h Makefile
        cc $(CFLAGS) -c neq.c

tank1.o: tank1.c tank1.h Makefile
        cc $(CFLAGS) $(DEFINES) -c tank1.c

basicTypes.o: basicTypes.c basicTypes.h Makefile
        cc $(CFLAGS) -c basicTypes.c
```

```
aList.o: aList.c aList.h Makefile
	cc $(CFLAGS) -c aList.c

driver.o: driver.c driver.h Makefile
	cc $(CFLAGS) -c driver.c

miniGun.o: miniGun.c miniGun.h Makefile
	cc $(CFLAGS) -c miniGun.c

commander.o: commander.c commander.h Makefile
	cc $(CFLAGS) -c commander.c

body.o: body.c body.h Makefile
	cc $(CFLAGS) -c body.c

driveTrain.o: driveTrain.c driveTrain.h Makefile
	cc $(CFLAGS) -c driveTrain.c

throttle.o: throttle.c throttle.h Makefile
	cc $(CFLAGS) -c throttle.c

brakes.o: brakes.c brakes.h Makefile
	cc $(CFLAGS) -c brakes.c

brakePedal.o: brakePedal.c brakePedal.h Makefile
	cc $(CFLAGS) -c brakePedal.c

steering.o: steering.c steering.h Makefile
	cc $(CFLAGS) -c steering.c

steeringLever.o: steeringLever.c steeringLever.h Makefile
	cc $(CFLAGS) -c steeringLever.c

fueltank.o: fueltank.c fueltank.h Makefile
	cc $(CFLAGS) -c fueltank.c

engine.o: engine.c engine.h Makefile
	cc $(CFLAGS) -c engine.c

driveshaft.o: driveshaft.c driveshaft.h Makefile
	cc $(CFLAGS) -c driveshaft.c

transmission.o: transmission.c  transmission.h Makefile
	cc $(CFLAGS) -c transmission.c
```

```
shifter.o: shifter.c shifter.h Makefile
	cc $(CFLAGS) -c shifter.c

clutch.o: clutch.c clutch.h Makefile
	cc $(CFLAGS) -c clutch.c

gearshifter.o: gearshifter.c gearshifter.h Makefile
	cc $(CFLAGS) -c gearshifter.c

treads.o: treads.c treads.h Makefile
	cc $(CFLAGS) -c treads.c

track.o: track.c track.h Makefile
	cc $(CFLAGS) -c track.c

poweredWheel.o: poweredWheel.c poweredWheel.h Makefile
	cc $(CFLAGS) -c poweredWheel.c

wheel.o: wheel.c wheel.h Makefile
	cc $(CFLAGS) -c wheel.c
```

Bibliography

1. "HITMISS." program.
2. "XPatch." program.
3. Apple, "PT 39 - The DR Emulator." Apple Computer. http://www.info.apple.com/dev/technotes/Platforms_&_Tools/pt_39.html.
4. Bain, William L. "Aggregate Distributed Objects for Distributed Memory Parallel Systems," *IEEE*, 1050–1055 (March 1990).
5. Bergman, Kenneth C. *Spatial Partitioning of a Battlefield Parallel Discrete Event Simulation*. MS thesis, AFIT/GCS/ENG/92D-03, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH 45433, December 1992. AD-A258911.
6. Booth, Guy R. *Implementation of an Object-Oriented Flight Simulator D. C. Electrical System on a Hypercube Architecture*. MS thesis, AFIT/GCE/ENS/91D-01, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH 45433, December 1991. AD-A243700.
7. Breeden, Thomas A. *Parallel Simulation of Structural VHDL Circuits on Intel Hypercubes*. MS thesis, AFIT/GCE/ENG/92D-01, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH 45433, December 1992. AD-A258999.
8. Brewer, Eric A. and William E. Weihl. "Developing Parallel Applications Using High-Performance Simulation." *Proceedings of the 1993 Workshop on Parallel and Distributed Debugging*. 1993.
9. Chandy, K. M. and J. Misra. "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs," *IEEE Transactions on Software Engineering*, *SE-5*(5):440–52 (September 1979).
10. Chandy, K. M. and J. Misra. "Asynchronous Distributed Simulation Via a Sequence of Parallel Computations," *Communications of the ACM*, *24*(11):198–206 (April 1981).
11. Chien, Andrew A. and William J. Dally. "Experience with Concurrent Aggregates (CA): Implementation and Programming," *IEEE*, 1040–1049 (March 1990).
12. DEC, "Alpha7 Desktop Supercomputer." http://www.ife.ee.ethz.ch/music/alpha/alpha.html. Digital Equipment Corporation.
13. DMSO, "ESAMS - Enhanced Surface-to-Air Missile Simulation." http://www.dmso.mil/mnscatalogs/afsaa/esams.html. Defense Modeling and Simulation Office.
14. DMSO, "TAC EC - Tactical Electronic Combat Systems." http://www.dmso.mil/mnscatalogs/afsaa/tacec.html. Defense Modeling and Simulation Office.
15. DMSO, "Thunder." http://www.dmso.mil/mnscatalogs/afsaa/thunder.html. Defense Modeling and Simulation Office.

16. Fujimoto, Richard M. "Parallel Discrete Event Simulation." *Proceedings of the 1989 Winter Simulation Conference*, edited by Edward A. MacNair and others. 19–29. December 1989.

17. Fujimoto, Richard M. "Parallel Discrete Event Simulation: Will the Field Survive?," *ORSA Journal on Computing*, 5(3):213–30 (Summer 1993).

18. Guanu, Seth R. *Dynamic Load Balancing for a Parallel Discrete-Event Battlefield Simulation*. MS thesis, AFIT/GCS/ENG/94D-25, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH 45433, January 1994. AD-A280666.

19. Hiller, James B. *Analytic Performance Models For Parallel Battlefield Simulation Using Conservative Processor Synchronization*. MS thesis, AFIT/GCS/ENG/94D-08, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH 45433, December 1994. AD-A289249.

20. Hurford, Joel F. *Accelerating Conservative Parallel Simulation of VHDL Circuits*. MS thesis, AFIT/GCS/ENG/94D-10, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH 45433, December 1994. AD-A289317.

21. Joint Modeling and Simulation System Program Office, ASC/XREM, WPAFB OH 45433. *Joint Modeling and Simulation System User's Guide*, 1994.

22. Kapp, Kevin L. *Partitioning Structural VHDL Circuits for Parallel Execution on Hypercubes*. MS thesis, AFIT/GCE/ENG/93D-07, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH 45433, December 1993. AD-A274390.

23. Kumar, Vipin and others. *Introduction to Parallel Computing - Design and Analysis of Algorithms*. Redwood City, California 94065: The Benjamin/Cummings Publishing Company, Inc., 1994.

24. Moser, Robert S. *A Spatially Partitioned Parallel Simulation of Colliding Objects*. MS thesis, AFIT/GCS/ENG/91D-15, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH 45433, December 1991. AD-A243967.

25. MSU, "Message-Passing Interface." http://www.erc.msstate.edu/mpi/. Mississippi State University.

26. Nicol, David M. "The Cost of Conservative Synchronization in Parallel Discrete Event Simulations," *Journal of the Association for Computing Machinery*, 40(2):304–333 (April 1993).

27. ORNL, "PVM: Parallel Virtual Machine." http://www.epm.ornl.gov/pvm/pvm_home.html. Oak Ridge National Laboratory.

28. Saphir, Bill and Sam Fineberg, "Performance Comparison of MPL, MPI, and PVMe." Argonne National Laboratory. http://lovelace.nas.nasa.gov/Parallel/SP2/MPIPerf/report.html.

29. Schuppe, Thomas F. "Modeling and Simulation: A Department of Defense Critical Technology." *Proceedings of the 1991 Winter Simulation Conference*, edited by

Barry A. Nelson and others. 519–25. 10662 Los Vaqueros Circle, PO Box 3014, Los Alamitos CA 90720-1264: IEEE Computer Society Press, December 1991.

30. Soderholm, Steven R. *A Hybrid Approach to Battlefield Parallel Discrete Event Simulation*. MS thesis, AFIT/GCS/ENG/91D-23, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH 45433, December 1991. AD-A24375.

31. Su, Wen-King and Charles L. Seitz. "Variants of the Chandy-Misra-Bryant Distributed Discrete-Event Simulation Algorithms." *Proceedings of the 1989 SCS Multiconference on Distributed Simulation*, edited by Brian Unger and Richard Fujimoto. 38–43. San Diego, CA: Society for Computer Simulation International, March 1989.

32. Trachsel, Walter Gordon. *Object Interaction in a Parallel Object-Oriented Discrete-Event Simulation*. MS thesis, AFIT/GCS/ENG/93D-22, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH 45433, December 1993. AD-A274084.

33. Walton, Andrew C. *Minimizing the Impact of Synchronization Overhead in Parallel Discrete Event Simulations*. MS thesis, AFIT/GCS/ENG/94D-25, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH 45433, December 1994.

34. Woolf, Henry B., editor. *Webster's New Collegiate Dictionary*. Springfield, MA: G C Merriam Company, 1979.

35. WPAFB, "Paragon Facility." WPAFB. http://msrc.wpafb.af.mil/ msrc/paragon.html.

36. Zeigler, Bernard P. "Hierarchical, modular discrete-event modelling in an object-oriented environment," *Simulation*, *49*(5):219–230 (November 1987).

37. Zeigler, Bernard P. *Object-Oriented Simulation with Hierarchical, Modular Models: Intelligent Agents and Endomorphic Systems*. 1250 Sixth Avenue, San Diego, CA 92101: Academic Press, Inc., 1990.

Vita

Capt Conrad P. Masshardt was born on 28 October 1967 in Madison, Wisconsin. He graduated from Oregon Senior High School in 1986 and entered undergraduate study at the University of Wisconsin-Madison. He graduated with a Bachelor of Science in Electrical Engineering in May 1991 and received his commission on 19 May 1991 as a Second Lieutenant.

Capt Masshardt's first assignment was at Wright-Patterson AFB as a development engineer in the Aeronautical Systems Center, where he worked on several projects including HARM integration on the F-15E, the Joint Modeling and Simulation System, as well as others.

Capt Masshardt was selected to attend the Air Force Institute of Technology in 1994.

Permanent address: 4723 Rome Corners Road
Brooklyn, WI 53521